API Testing

- API is a Application program interface and it works as a business logic b/w presentation layer and database layer

There are 3 types of application architecture:

  - 1 Tier: here client(browser/computer/mobile) and server are present connected in a same local machine
  - 2 Tier: here clients are connected to a database server which is a common severe for multiple client and its hosted somewhere else
  - 3 Tier: here client is connected to business logic and this in turn is connected to database layer
- API follows 3 tier app architecture. Developers develop APIs which acts as a business logic and it's the intermediator b/w frontend layer and backend layer
- API testing is also considered as Backend testing
- Developers first setup database and the APIs are developed and on top of this Web app is developed
- Functional testing is done through API testing. Around 70-80% of testing can be doe through API testing and rest can be done as a UI testing on frontend

There are 2 types of APIs:

  - SOAP – Simple Object Access protocol
  - REST – Representational State Transfer
- SOAP is one of the old API type which uses xml for API creation. REST is the latest one widely used as it supports xml, json, excel, etc
- Both types are **webservices**

What is Webservice:

- All Webservices are APIs but not all APIs are webservices, during application development phase the APIs created are considered as the APIs once these APIs are deployed to production that means once theses are available for public to access in the internet then these APIs are called as webservices. All the APIs available in internet for public use such as Google map APIs, Google account APIs, etc are called Webservices

- Webservices are the APIs wrapped in http

- Webservices needs network whereas APIs do not need network for its operation

REST API http methods:

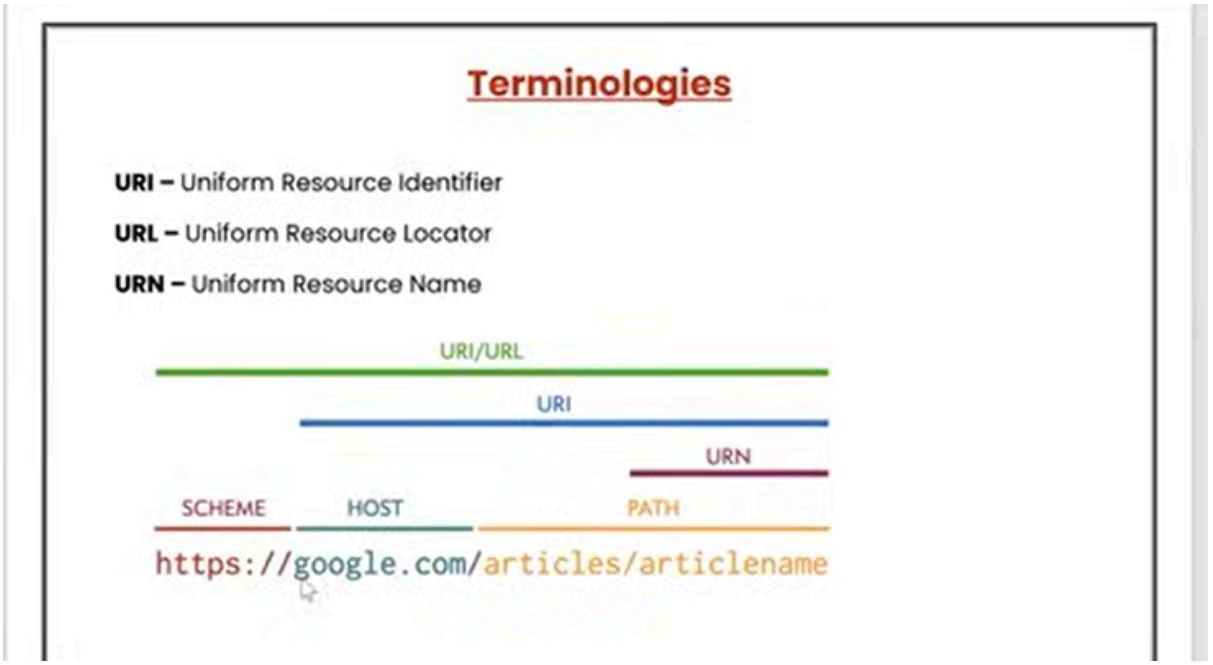
- REST APIs are widely used because of their methods out of which 4 are the main ones:
  - GET: retrieve resource(functionality) from server

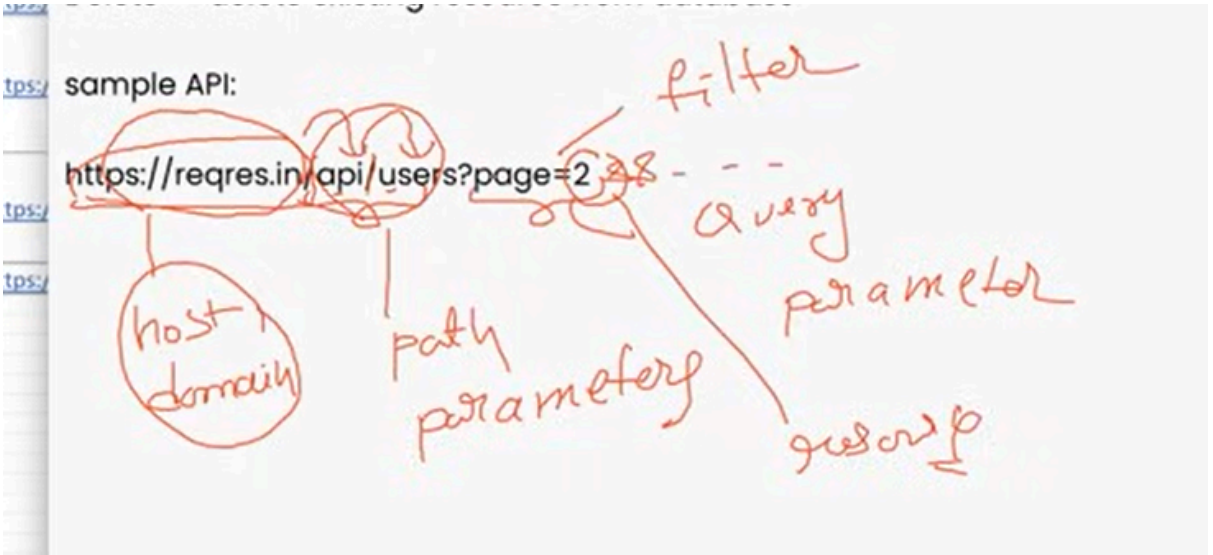
    - POST – create a nre resource in server
    - PUT – update an existing resource completely
    - DELETE - delete an existing resource completely
    - PATCH - update an existing resource partially
- Any of these are used all the time for every transactions made in web app.

HTTP vs HTTPS

- Any request sent with http protocol the data sent will be in original format to the server whereas in case of HTTPS the data sent will be in encrypted format to the server hence it is more secure and chances of getting hacked is less compared to HTTP

URI, URL, URN



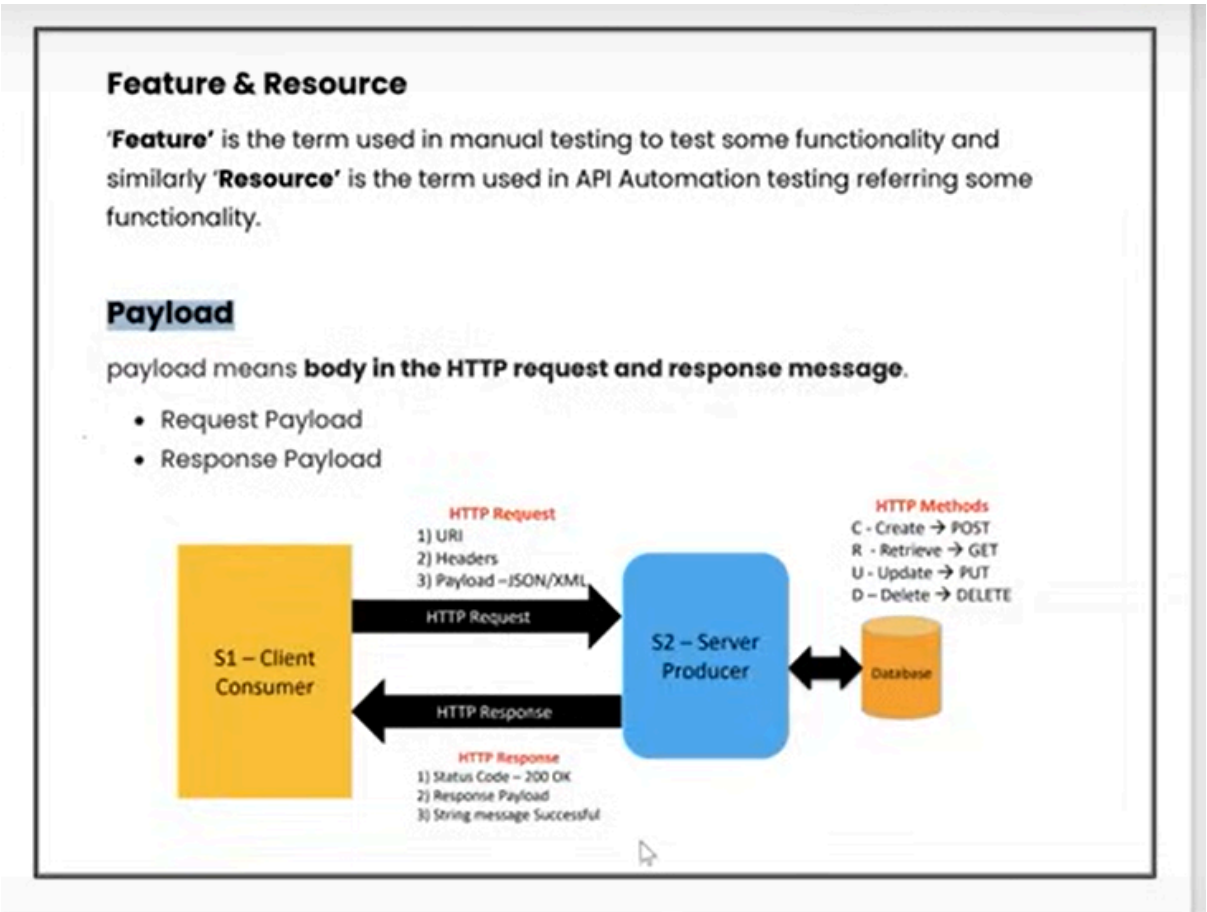


Feature and Resource

- In manual testing feature is termed to the functionalities to be tested whereas in API testing Resource is termed to the functionalities to be tested

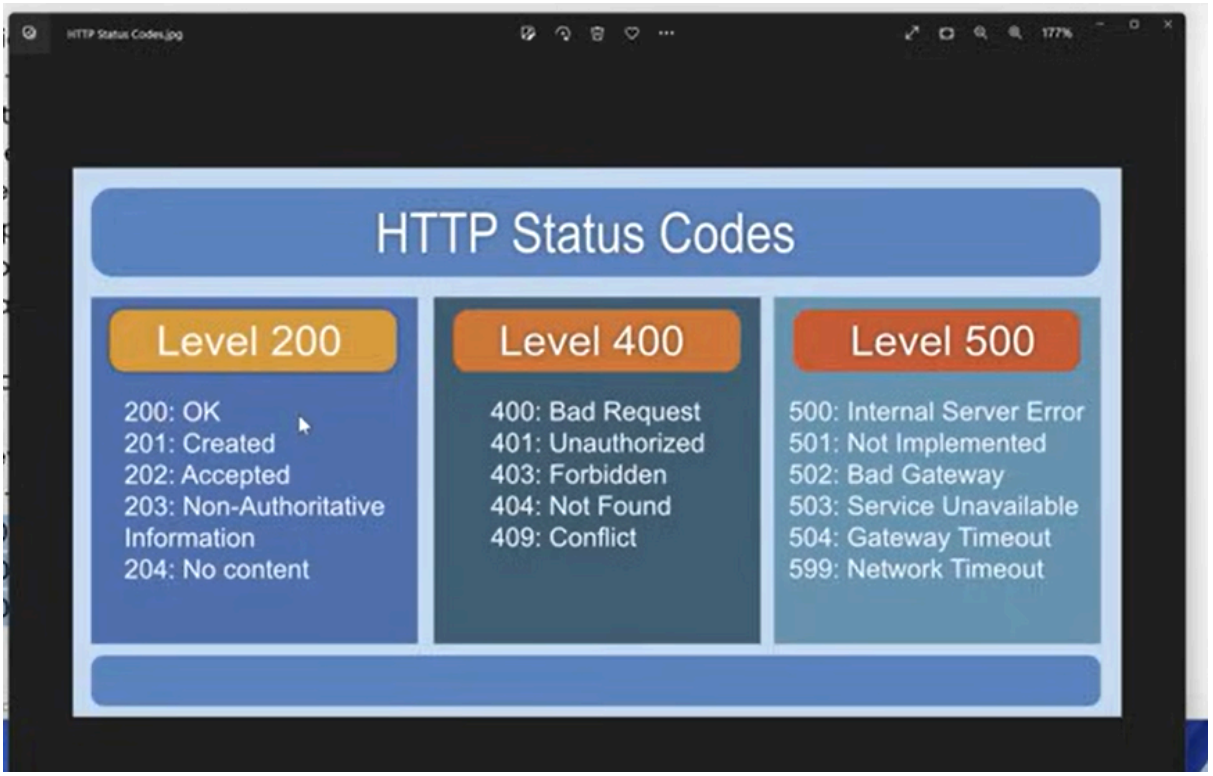
Payload

- Any data that user sends with the request to the API these data are considered as Request Payload and a data sent in the response by the API are considered as Response Payload



- During HTTP request that user send it contains mainly URI, Request payload in JSON/XML and headers and the response received contains Response payload, status code and string message

HTTP Status Codes:



Creating our own APIs for testing

- We need 2 tools:
  - Install NodeJS
    - It should also contain npm – node package manager
  - Install json-server
- After installing nodeJS setup the nodejs path in environment variable

  C:\Program Files\nodejs

  C:\Program Files\nodejs\node_modules

- Now check if nodejs is installed by checking its version in cmd:
  - node --version
- Now check npm is also installed by checking its version in cmd:
  - npm –version
- Now we need to install json-server: for this run this command in cmd:
  - npm install -g json-server
- With this we have installed 2 tools which will allow us to create our own APIs

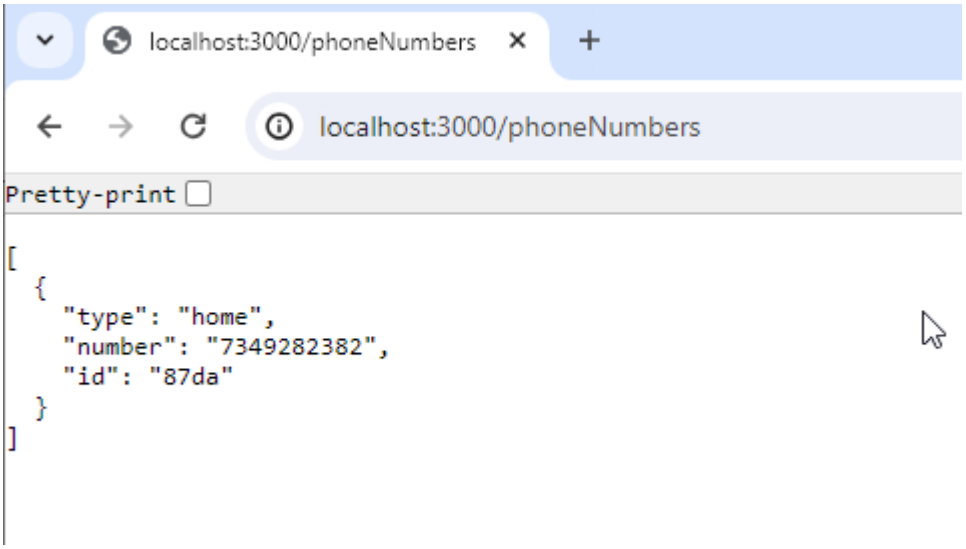
Create API

- Now we need a json file which will have the data which the API will use, create a simple json with some data or download a sample

sample.json

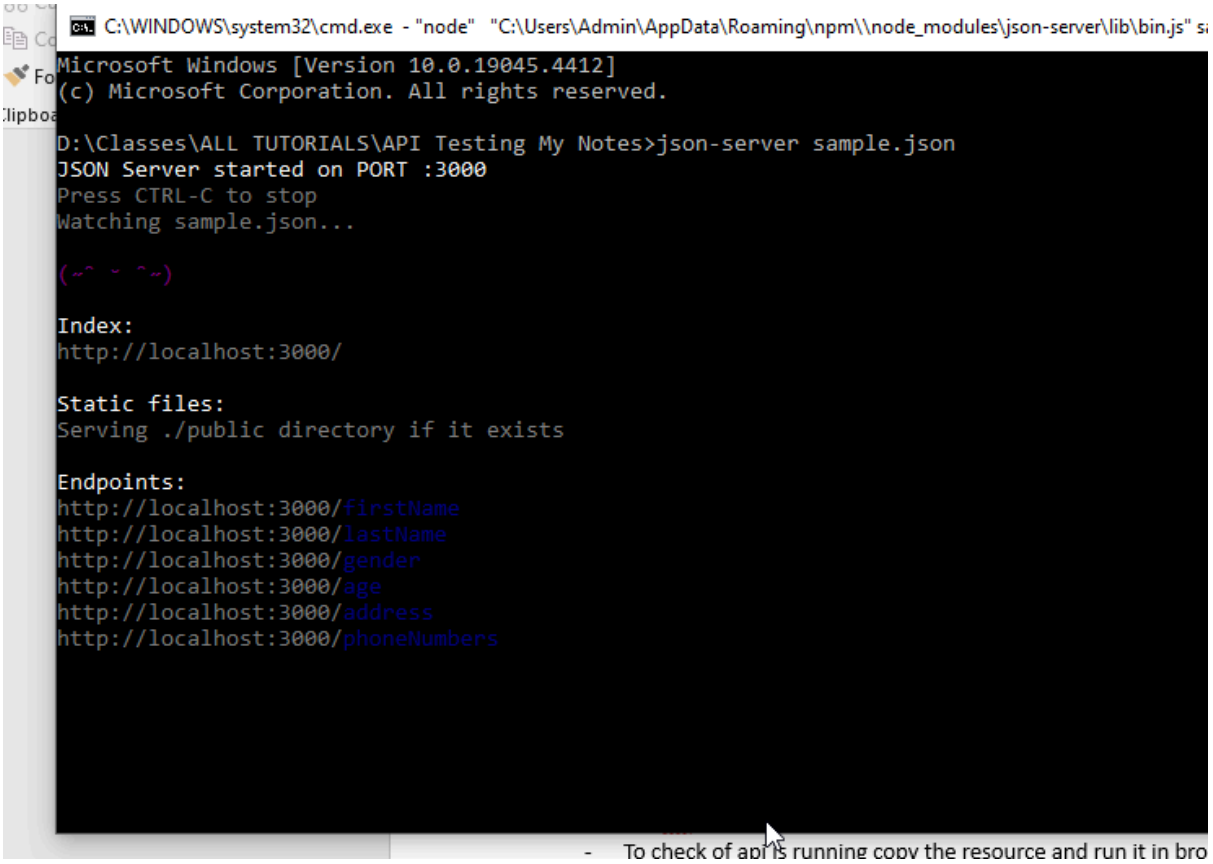
```json
{
    "firstName": "Joe",
    "lastName": "Jackson",
    "gender": "male",
    "age": 28,
    "address": {
        "streetAddress": "101",
        "city": "San Diego",
        "state": "CA"
    },
    "phoneNumbers": [
        { "type": "home", "number": "7349282382" }
    ]
}
```

- Now open the cmd from the path where the json file is stored
- Run the json file in cmd with command:
    - json-server <jsonFileName.json>
- With this the json file's API is up and running in the local machine. It will be terminated when the cmd is closed and to rerun once again run json in cmd and do not close the terminal
- We wil get resources/end points info in the terminal which will be the url
- To check of api is running copy the resource and run it in browser and the browser will fetch the json data



- So what is happening is:
    - client that is browser is sending an API request(end points of json file in terminal)
    - API sending the request to the server in this case its local m/c
    - Searching the data from the home/host that is the json file
    - And sending back the expected data from the file to the client as a response

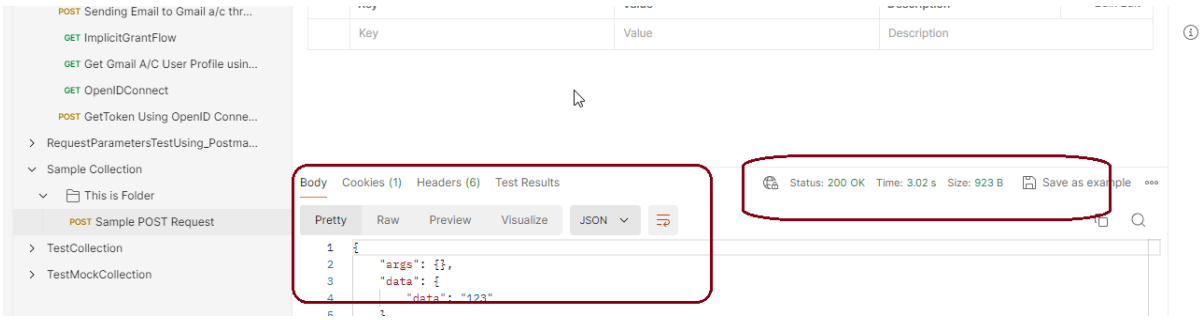
- This Is the way to create our own dummy APIs and we can run the same APIs in Postman



- To check of api is running copy the resource and run it in bro

What can be validated in API request

- Following can be validated in http response once an http request is sent
  - Response body
  - Cookies
  - Headers
  - Status code
  - Time taken
  - Size
- All these will be available in postman

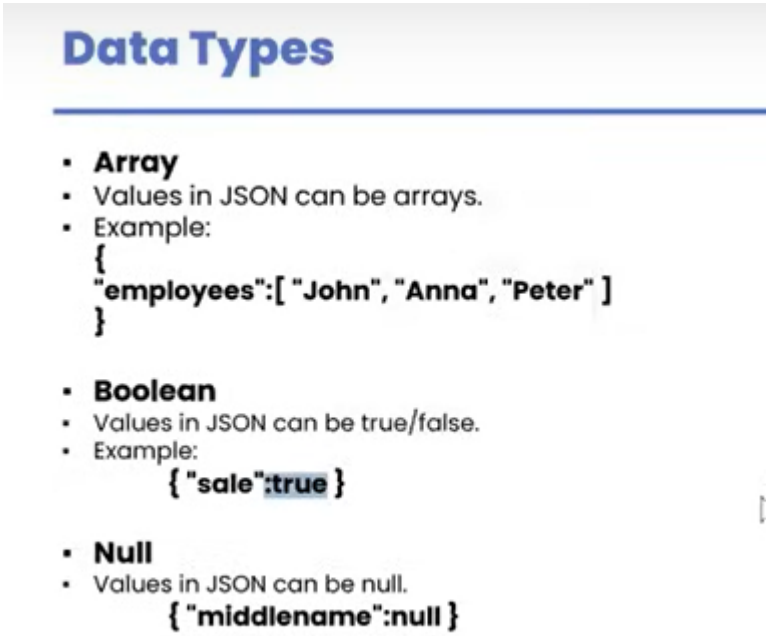


JSON/XML

- JSON is called Java Script Object Notation and it is a Data format just like pdf, html, excel, etc
- XML is called Extensible Markup Language and it is a Data format just like pdf, html, excel, etc
- JSON or xml format is always used to write the request body and send to server and the response body is also received in json or xml format
- These are widely used in API testing and JSON is widely used in REST API while XML is widely used in SOAP API
- These are used because they are light weighted that is the size is smaller compared to other formats and this allows faster request and response transactions
- These are also flexible for encryption and decryption hence allowing data protection

JSON Data types

- Json files will have **.json** extension
- The internet media type of json is **application/json** this we can see in header. This is also called as MIME type which defines the format of the data
- Json is always written in **key-value** pairs
- Following datatypes are used for Json data creation:
    - Number
    - String
    - Boolean
    - Null
    - Object
    - Array



{

“firstname”:”John”, - String

“age”:30, - integer

“phone”:[123, 12345], - array of integer

“status”:true, - boolean

“var”:null - null

}

- From { till } it is considered one object hence above data is a json object which contains different data types
- Lets say we want to create a json object containing 2 students data

```
{
“students”:
        [
                {
                “name”: “john”,
                “class”: 10
                },
                {
                “name”: “doe”,
                “class”: 11
                }
        ]
}
```

- Hence a students object is created and it has 2 student’s object inside in an array

JSON vs XML

JSON	XML
JSON is simple to read and write.	XML is less simple as compared to JSON.
It also supports **array**.	It doesn't support array.
JSON files are more **human-readable** than XML.	XML files are **less human readable**.
It supports only **text** and **number** data type	It supports many data types such as **text, number, images, charts, graphs**, etc.

Capture and Validate JSON Path

- Just like xpath we use to identify an elemnt in the xml we use JSON path to identify the data in the json

```
{
"students":
        [
                {
                "name": "john",
                "class": 10
                },
                {
                "name": "doe",
                "class": 11
                }
        ]
}
```

- To get name – john: **students[0].name =** This will return john
- To get class – 11: **students[1].class =** This will return 11
- There will be complex jsons involved in project to find the path in such json files we use json path finder tools such as:
    - https://jsonpath.com/ **- here we can check our json path is identifying the data correctly**
    - https://jsonpathfinder.com/ **- here we can generate a json path easily**

Response Validation (Adding tests)

- Following are validated:
    - Status code
    - Headers
    - Cookies
    - Response time
    - Response body
- In Json there are different assertion functions and these are available in library – **pm** Which is added in the postman.
- Pm library contains javascript functions which are used for assertions

Chai Assertion Library

- This is one of the javascript assertion library present in pm library with this we can add assertions to validate elements mention above
- The way its written is:

```
pm.test("Test Name", function()
                        {
                        // assertions are written here
                        } );
```

Above code can also be written like this using arrow => instead of function text

```
pm.test("Test Name", () =>
                        {
                        // assertions are written here
                        } );
```

- pm is the postman library
- .test is the function of pm library
- "Test name" – we add the nae of the test
- Function() it's the method/function inside which assertions are added. In place of this we can use =>
- There are 2 types os scripts user can add in postman
    - **Pre-Request scripts**: These will be executed as soon as user hits send and before the request is sent to the server. These scripts will validate the request and its data
    - **Post-Response scripts:** These wil be executed once the response is generated. These scripts will validate the response and its data

Validation scripts defined below are Post- Response scripts

Testing Status code:

Testing status codes

Test for the response status code:

```
pm.test("Status code is 200", () => {
  pm.response.to.have.status(200);
});
```

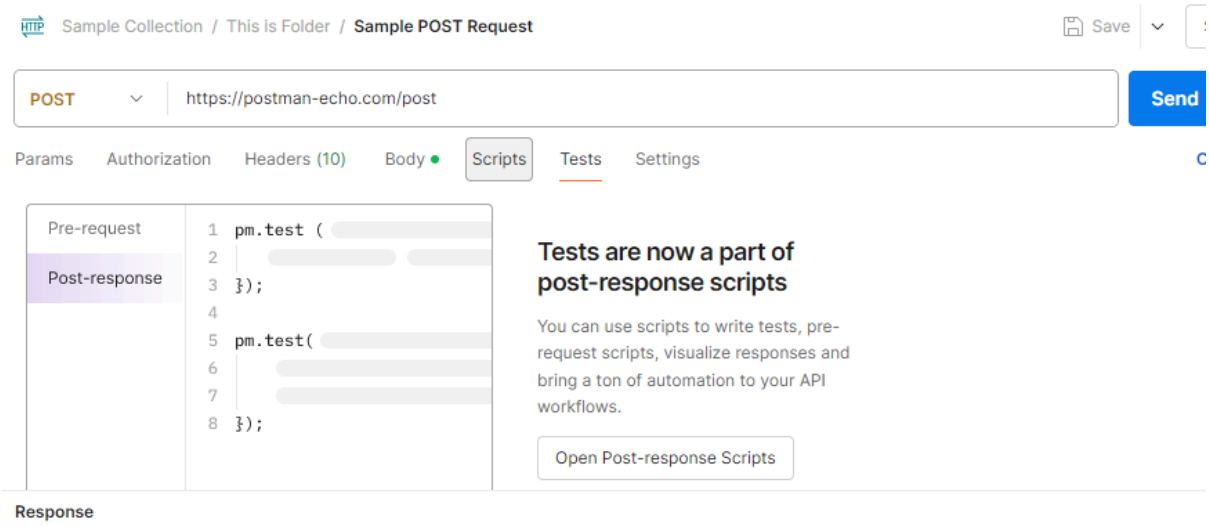
If you want to test for the status code being one of a set, include them all in an array and use *one of*

```
pm.test("Successful POST request", () => {
pm.expect(pm.response.code).to.be.oneOf([201,202]);
});
```

Check the status code text:

```
pm.test("Status code name has string", () => {
  pm.response.to.have.status("Created");
});
```

- These validation functions are added in the Test section of a request in postman



Validating response status code:

```
pm.test("test status code", () => {
        pm.response.to.have.status(200);
});
```

- We can write any description in test name section hence we wrote – **"test status code",**
- **response.to.have.status(int)** are the methods of pm library

Validating different Status codes in the response:

- There may be case when the response may send different status codes such as 200. 201. 202, etc and we want to validate that any of these status codes are returned instead of validating a single code we can use below approach:

```
pm.test("Test any status code", () => {
        pm.expect(pm.response.code).to.be.oneof([200,201,202]);
});
```

- Expect is a method in which we can write logic what to expect in this case we expect response code and to.be.oneof([int1,int2,int3]) here we add the values as arrays and with this it will validate the response code contains any one of the values added in the array

Validating response status code text:

```
pm.test("test status code", () => {
        pm.response.to.have.status("Created");
});
```

- Here we are validating the response status code text by passing string value to status method

Validating Response headers

- **Validating the presence of header key:**

```
pm.test("validate header is present", () => {
    pm.response.to.have.header("Content-Type");

    });
```

- **Validating the value of response header**

```
pm.test("validate header value", () => {
    pm.expect(pm.response.headers.get('Content-Type')).to.eql('appl
    ication/json');
    });
```

Validating Response Cookies

- **Validate cookie is present**

```
pm.test("validate presence of cookie", () => {
        pm.expect(pm.cookies.has('language')).to.be.true;
    });
```

- **Validate cookie value**

```
pm.test("validate cookie value", () => {
        pm.expect(pm.cookies.get('language')).to.eql('en-gb');
    });
```

Validating Response Time

- Response time always varies hence we may not be able to validate exact time, so the logic provided is to validate the response time is below the time defined in the code and the time defined in the code can behave as the max time limit within which the request must render the response

```
pm.test("validate response time", () => {
        pm.expect(pm.response.responseTime).to.be.below(30);
    });
```

- Here 30 is 30 ms which is the max response time user has defined

Validating Response Body

- **Validating the Data type stored by the fields/key in the json body**

- We need to create an object of javascript which will store the json response and then we get data from the object and validate expected info

```
// This is the object of const type which will store json response
const jsonData = pm.response.json();

pm.test("validate data type of fields in response body", () => {
    pm.expect(jsonData).to.be.an("object");
    pm.expect(jsonData.streetAddress).to.be.a("string");
    pm.expect(jsonData.number).to.be.a("number");
    pm.expect(jsonData.states).to.be.an("array");
});
```

- We can instantiate jsonData object inside pm.test but it will become local to this script, since we have instantiated outside it will behave as a global variable and we can use it in multiple scripts

Note: In Java script numbers data type is called as integer as well as number

- **Validating the Array in the json body**

{

“firstname”:”John”,

“age”:30,

“phone”:[123, 12345]

}

```
// This is the object of const type which will store json response
const jsonData = pm.response.json();

pm.test("validate array in response body", () => {
```

```
// Validates the array contains the data defined in bracket
// include method will take only single param to validate
 pm.expect(jsonData.phone).to.include("123");
// Validates the array contains the list data defined in bracket
// members method validate the entire list
    pm.expect(jsonData.phone).to.have.members(["123",12345]);
});
```

- **Validating the fields/value in json body**

```
// This is the object of const type which will store json response
const jsonData = pm.response.json();

pm.test("validate fields in response body", () => {
    pm.expect(jsonData. firstname).to.eql("John");
    pm.expect(jsonData. age).to.eql(30);
    pm.expect(jsonData. phone[0]).to.eql(123);
    pm.expect(jsonData. phone[1]).to.eql(12345);
});
```

Note: In all the examples the param of expect: jsonData.age, etc are the json path of the element present in the json body. If the json body is huge/complex we can define the complex json path in the expect param to locate the element

JSON Schema

- This is a document which defines the expected json response amd request body that means it defines what fields/properties/keys are expected and what data-type they will hold and also it will define what all fields are requires that is mandatory. It will be written in a json format with expected fields mentioned and their value's data-types
- If the schema is not followed by json body then it will throw error.

JSON schema

```
var schema={
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer"
    },
    "name": {
      "type": "string"
    },
    "location": {
      "type": "string"
    },
    "phone": {
      "type": "string"
    },
    "courses": {
      "type": "array",
      "items": [
        {
          "type": "string"
```

- Every json response and request generated will follow a json schema defined by the developer.
- Testers validate the json response is following the expected json schema are not.

Json schema for below json data:

```
{
  "firstName": "Joe",
  "age": 28,
  "address": {
    "state": "CA"
  },
  "phoneNumbers": [
    { "type": "home"}
]}
```

Json schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string"
    },
    "age": {
      "type": "integer"
    },
    "address": {
      "type": "object",
      "properties": {
        "state": {
          "type": "string"
        }
      },
      "required": [
        "state"
      ]
    },
    "phoneNumbers": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "type": {
              "type": "string"
            }
          },
          "required": [
            "type"
```

```
            ]
          }
        ]
      }
    },
    "required": [
      "firstName",
      "age",
      "address",
      "phoneNumbers"
    ]
}
```

JSON Schema Validation

- For schema validation we use 2 things:
  - **Json schema** of the response body
  - **tv4 library** which is called Tiny Validator version 4 and it is used for schema validation
- First we create a variable to store json data and the we create a variable to store json schema and the use usual code and add them in Test field of postman:

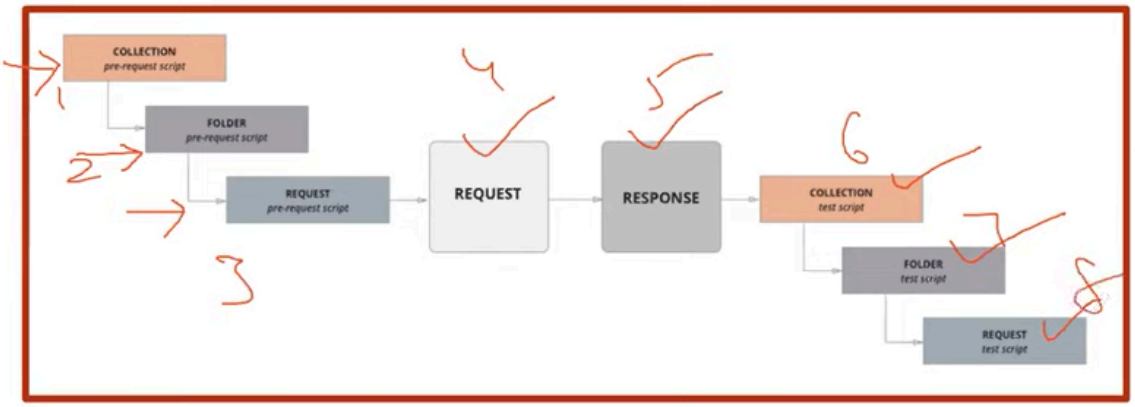
```
const jsonData = pm.response.json();
```

```
// variable storing schema
var schema = {json schema added here}
```

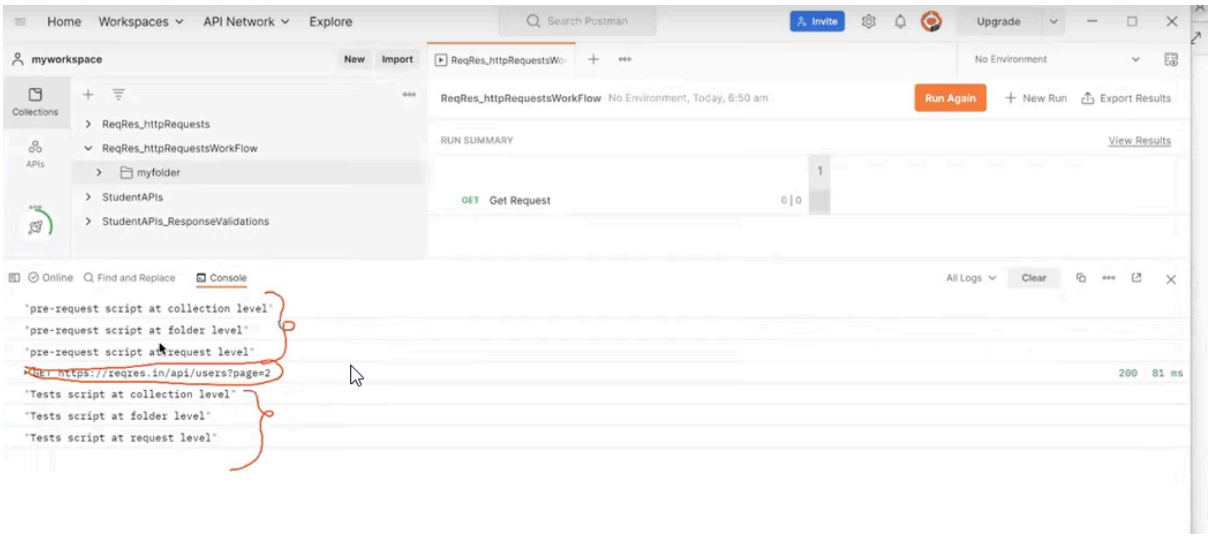
```
pm.test("json schema validation", () => {
    pm.expect(tv4.validate(jsonData, schema)).to.be.true;
} );
```

Note: All the validation scripts defined above are Post-Response scripts

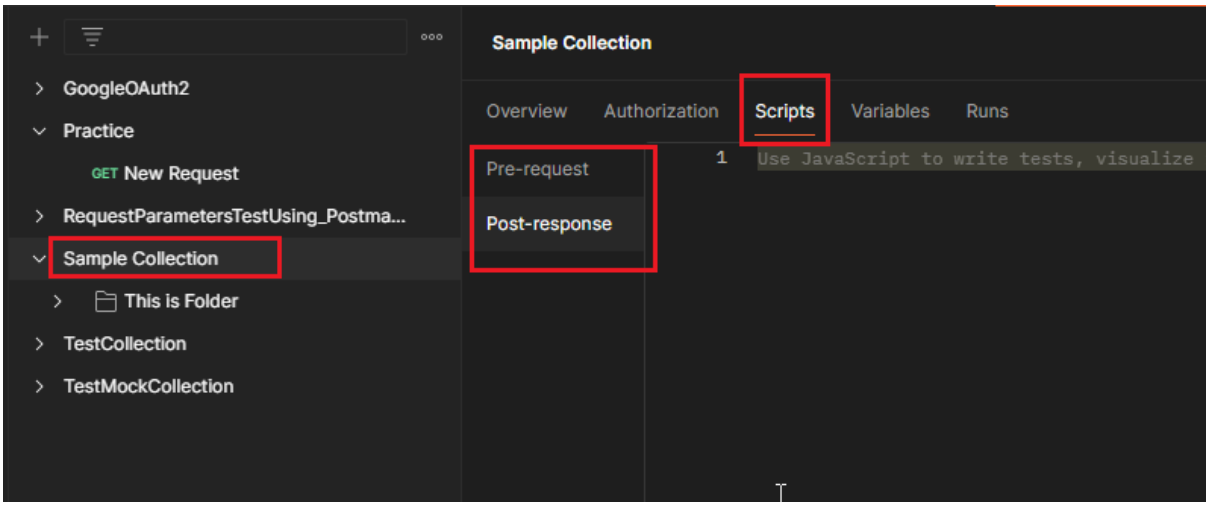
- **Pre Request and Post response scripts** can be added at following levels:
  - Collection level
  - Folder level
  - Request level
- **Sequence of script execution will be:**
  - 1st Collection's Pre Request will execute before request is sent and then 1st Collection's Post response scripts will execute ater the response is generated
  - 2nd folder level will be executed
  - Finally script level pre and post request will be executed



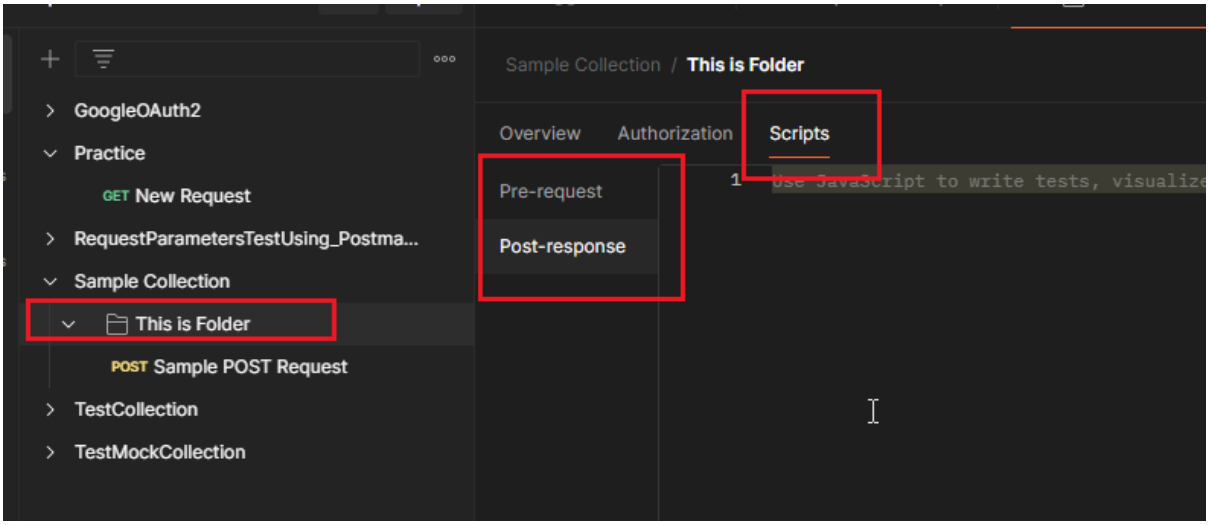
Scripts executed at different level displayed in postman console below



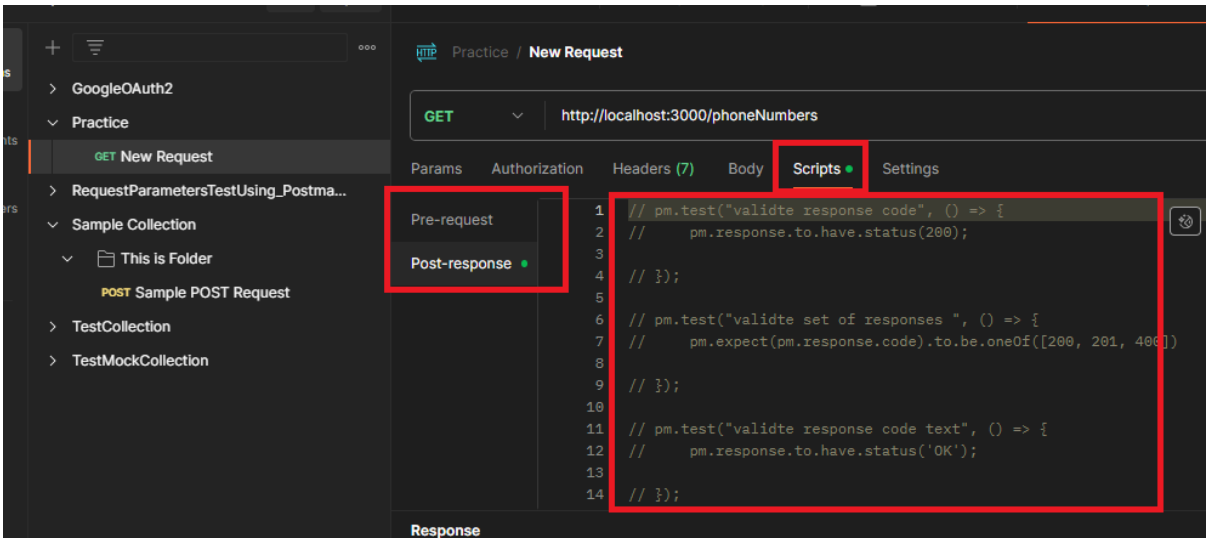
- **Collection level**



- **Folder level**



- **Request level**



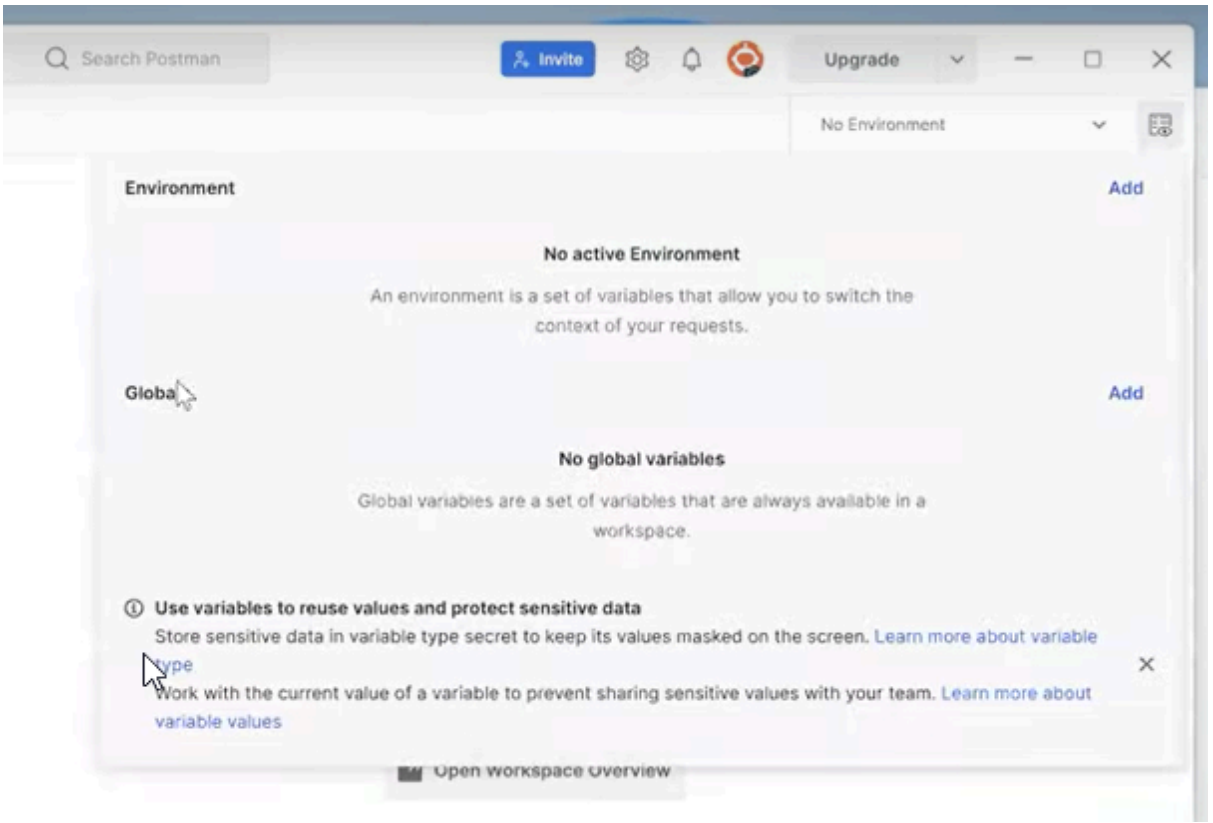
- Pre-Request scripts are used mainly for 2 reasons:
  - Whenever we are sending a POST request we usually send a data in the body to the server and this POST request we may send multiple times but the issu arise wil be we don't want to post duplicate data in server so we change the field values in boy and send post request, imagine creating data entries for each students in the server. Now manually updating the body and sending the request can lead to time consumption and extra workload, this can be handled by pre-request scripts. We can create a script which will generate random values for the fields and generate values with some logic and store these data in the body and send it in the post request
  - We can also send parameters to the sever through the pre-request scripts
- Pre and post request scripts defined at collection level will work on collection, folder and request levels. Pre and post scripts defined at folder level will work on folder and request levels, Pre and post scripts defined at request level will work on request level only

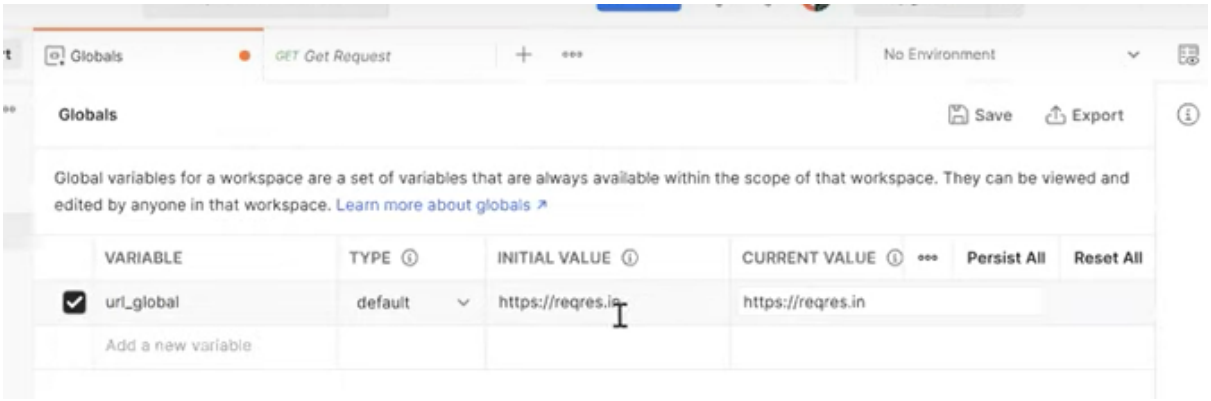
Variables

- Postman supports different types of variables to store data such as url, params, etc
- It is useful to avoid duplication, redundant data, rework of updating data multiple places, using global variable across all test scripts
- Based on Scope of accessibility of variables there are 5 types/levels of variable:
    - **Global variables**
    - **Collection variables**
    - **Environment variables**
    - **Local variables**
    - **Data variables**

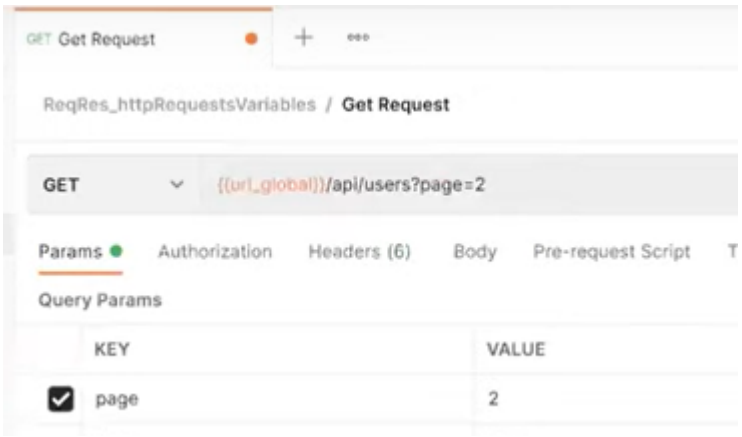
- **Global variable:**

- It is defined in environment setup section
- These variables will be accessible throughout the workspace that is we can use it at all collections, folders and request levels

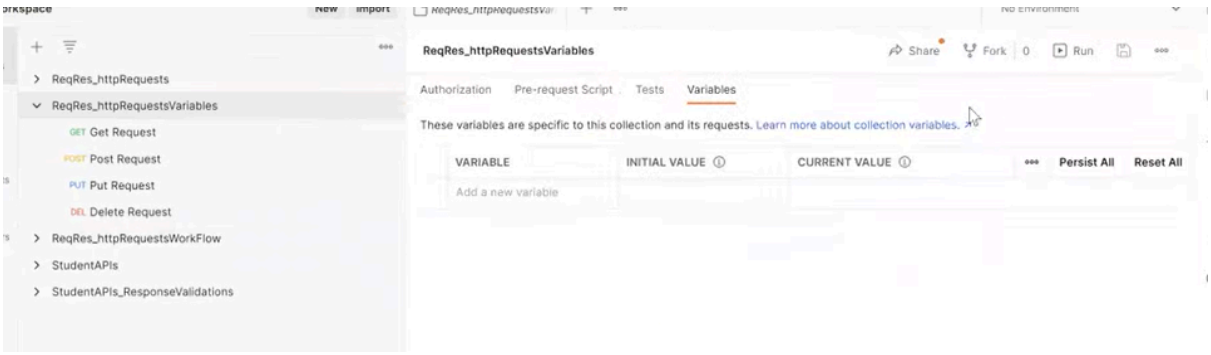




- Url is made as global variable: url_global
- We can use this variable in all the request urls like this: **{{url_global}}/uri**
- We can use these variables in all the places and format to use will follow: **{{variableName}}**

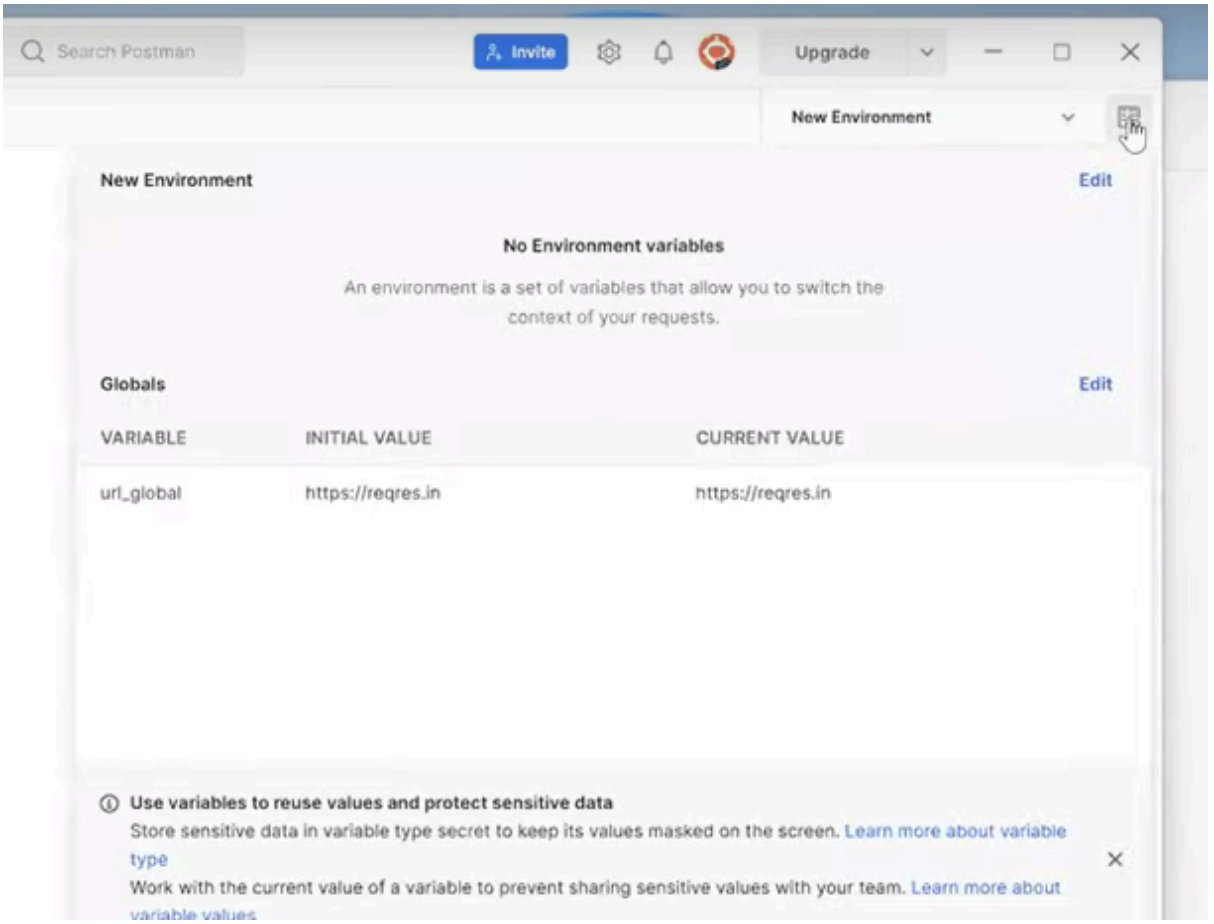


- **Collection variables**
- These are defined at collection levels and these will be accessible with the defined collection that means the collection's folder and all http requests can access it but other collections and their requests can't use it
- Click on edit link of collections and in edit page we can find variables tab
- We can use these variables in all the places within the collection and format to use will follow: **{{variableName}}**

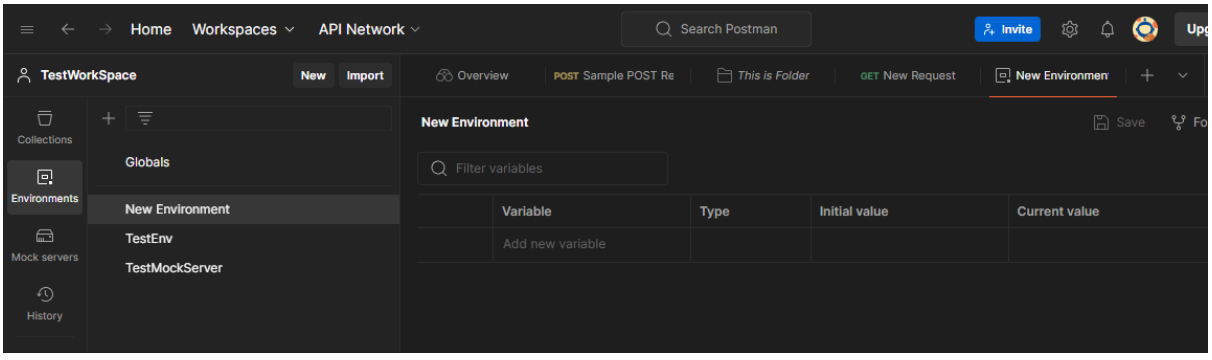


- **Environment variables**

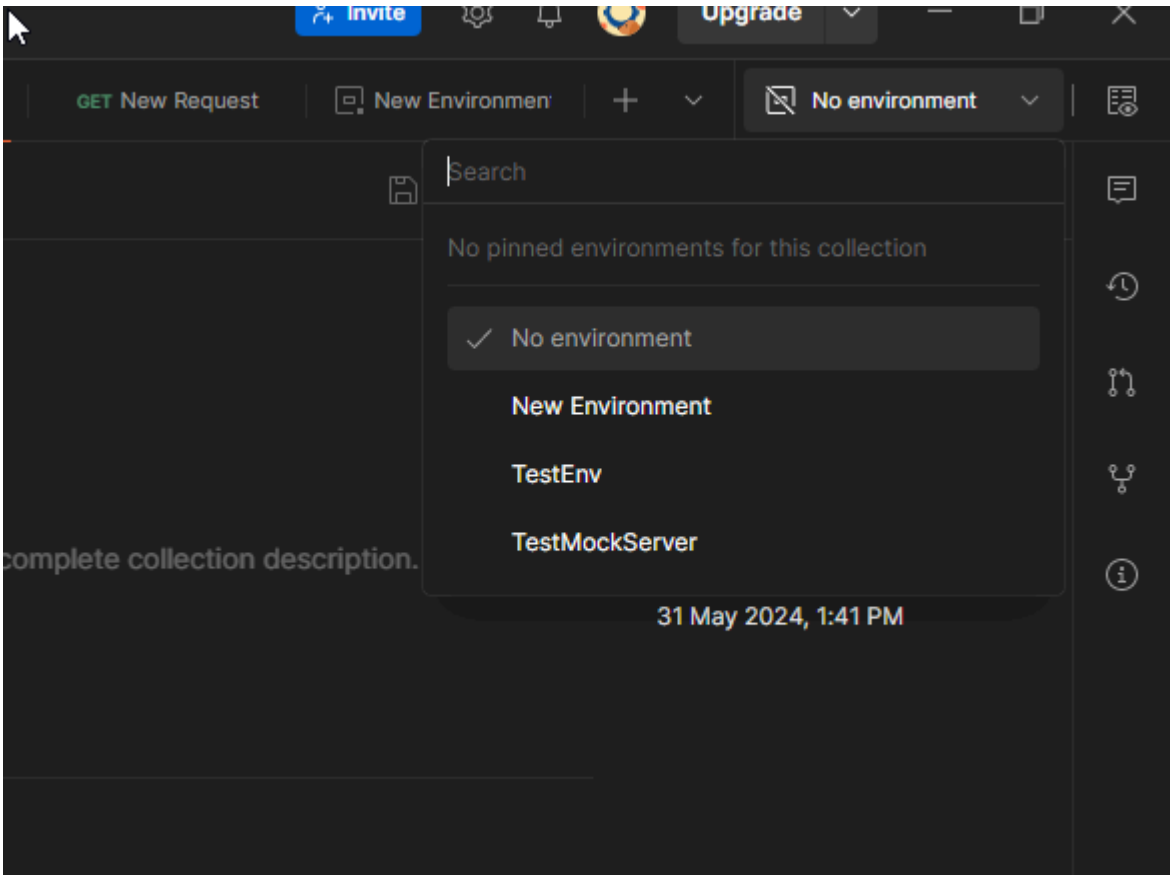
- These are also similar to globa; vars but the main usage involves different envs and their own variables. Lets say we want to test a collection with multiple requests in QA, DEV and Stage env, These envs will have there own urls, test data/vars, hence we can create an environment ariable for each test envs and these envs will contain their own test vars and we can switch between the env vars to test the same collection
- We can create it inside env setup option



- Also we have a Environment tab at left pane



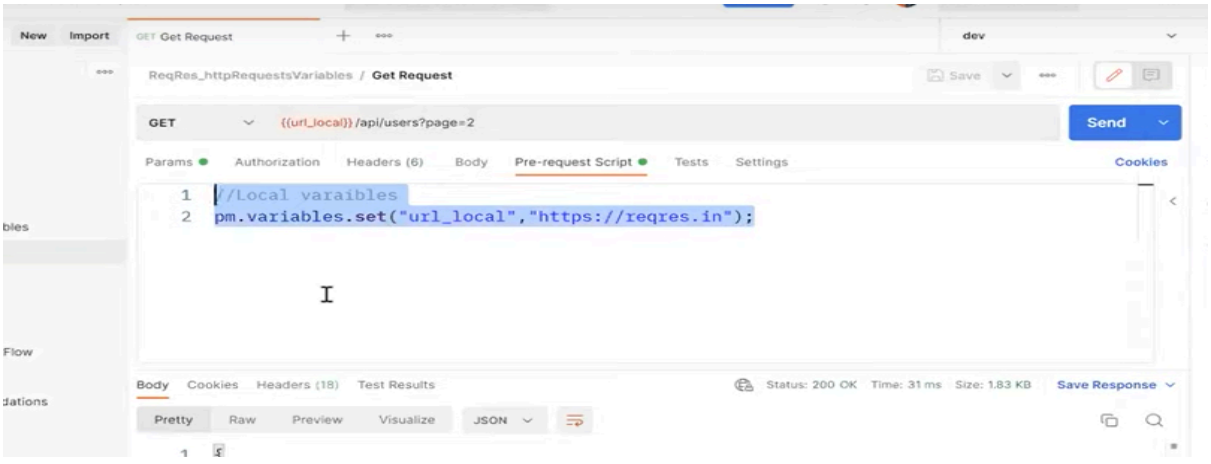
- Once env var is created we need to switch to that env from top right corner option in order to access its var in all the collection, folder and requests.



- If we do not switch to env variable and try to use its vars then it will throw error and this is a main difference b/w global and env var, where global var can be used anywhere anytime whereas env var can be used only after switch to that var

- **Local Variables**
- These are defined at request level
- These wil be accessible only for the request in which its defined
- Its added as a script in pre-request script body
- Pm library is used for defining local variable:

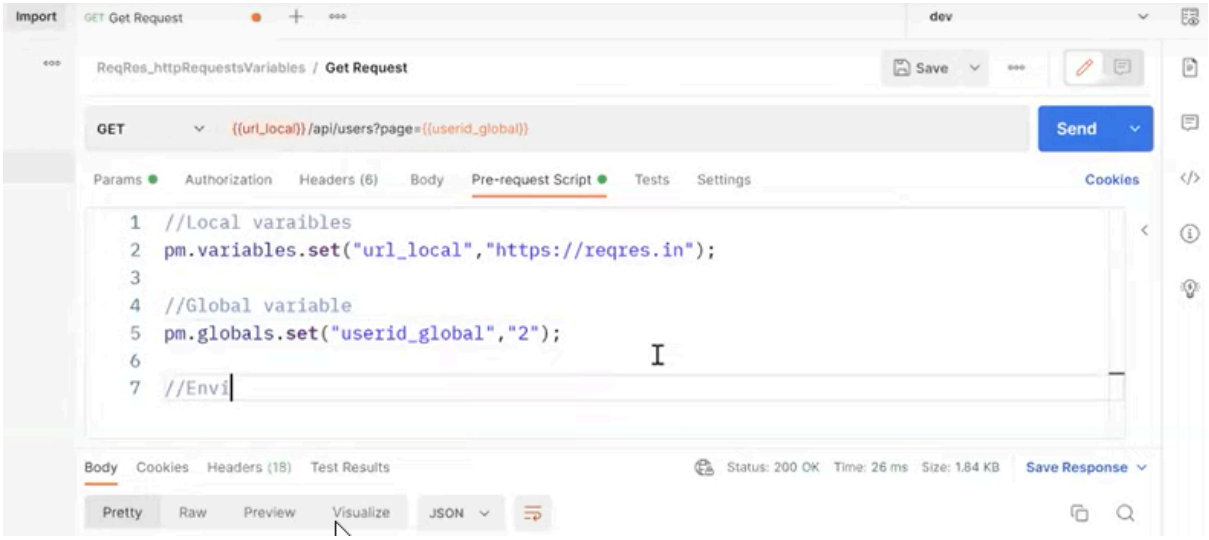
pm.variables.set("key","value");



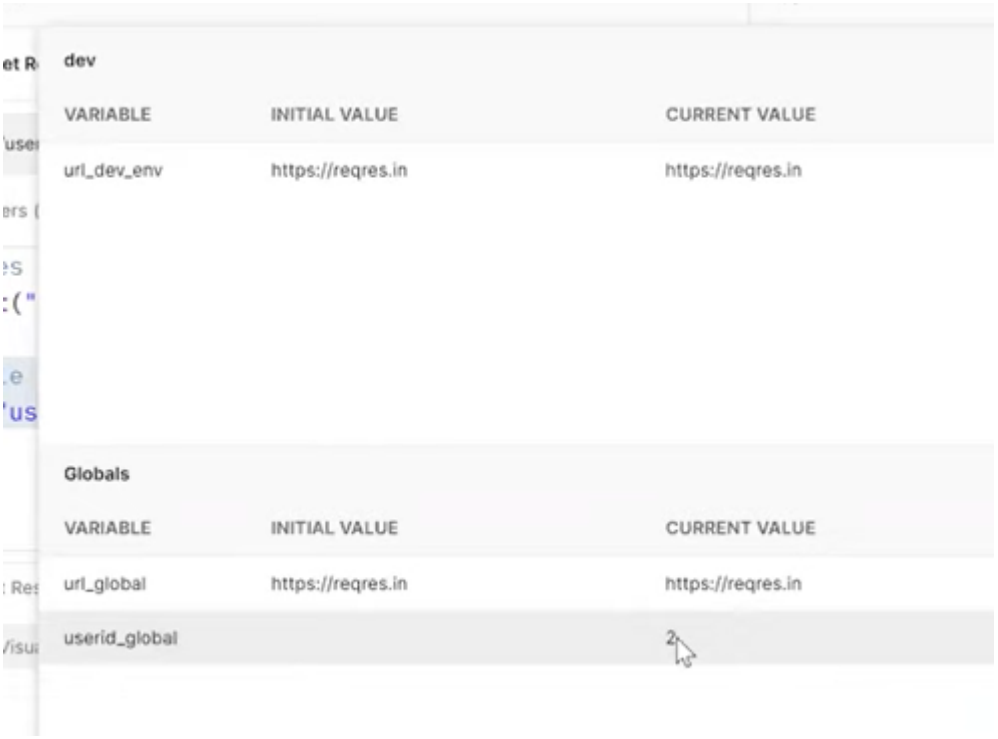
- **Data variables: U**se of external file such as csv file, text file etc. It is also considered as DDT.

- We can also create Global, environment and collection variables in the pre-request script just like local variable.

Global variable defined in pre-request script:

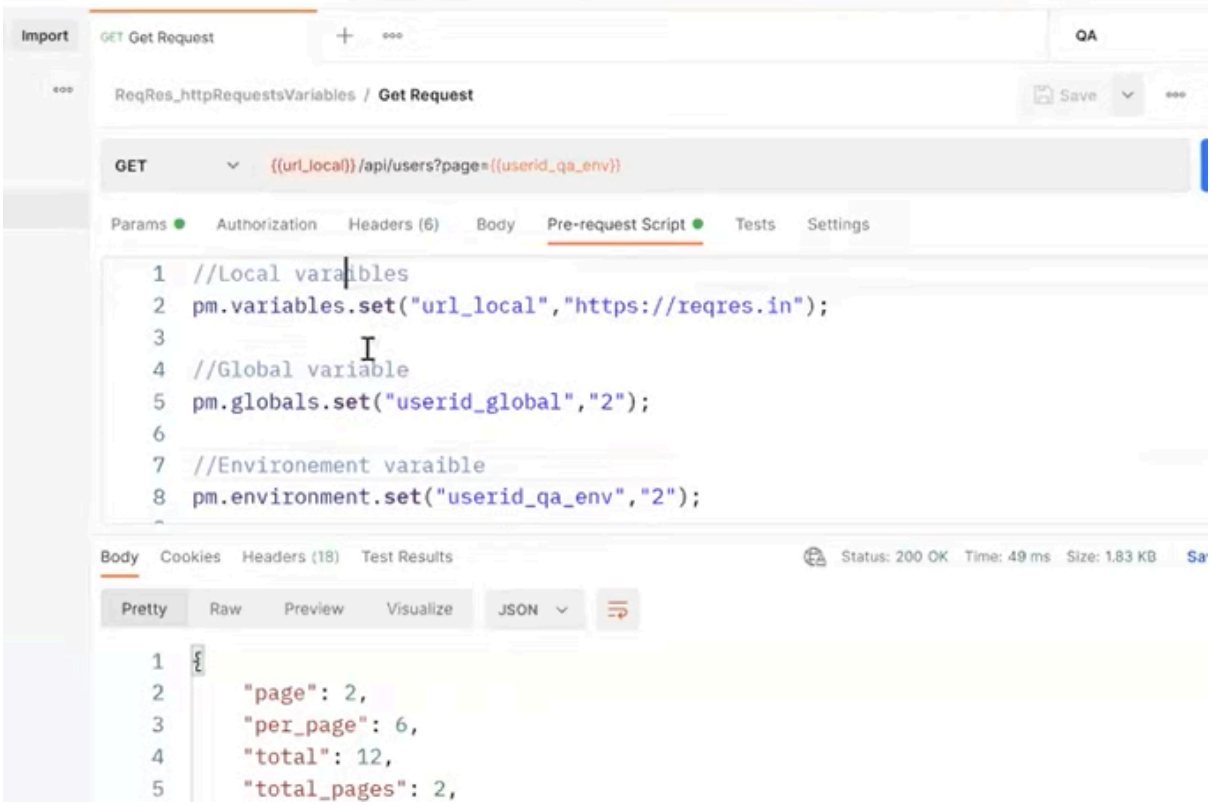


- These variables when defined in pre-request scripts they get added in the variable setup level during execution

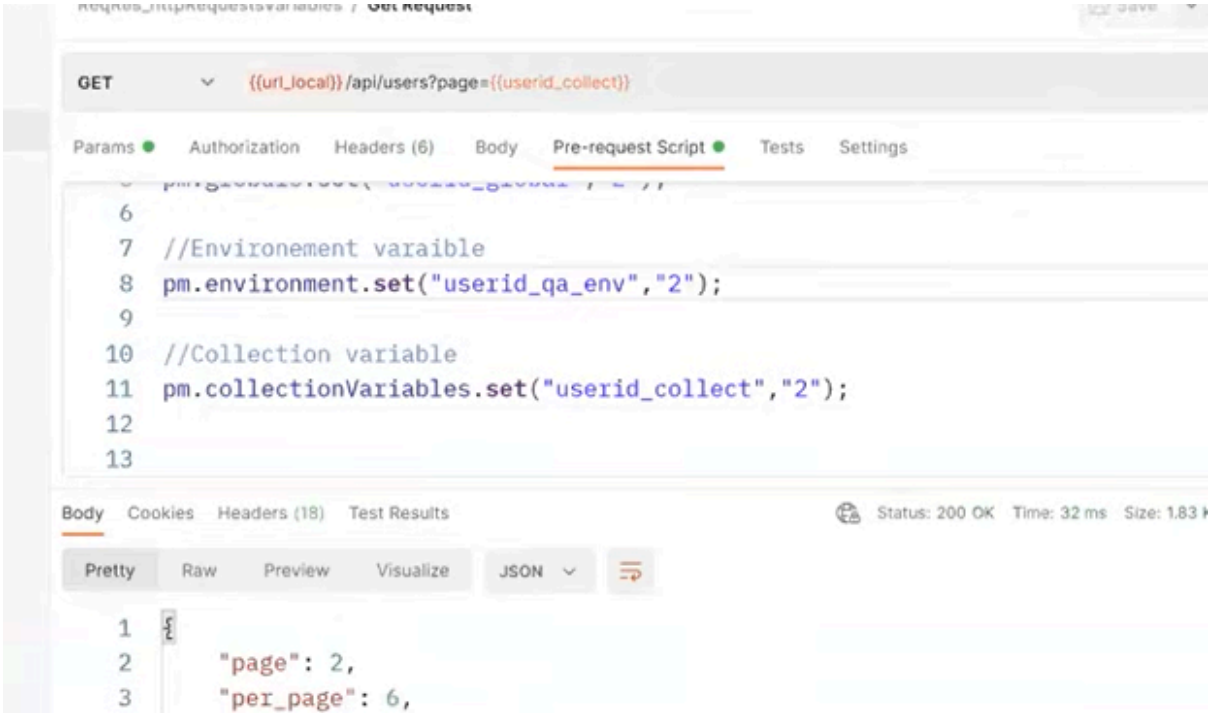


- These variables can be used in the request url before sending the request – {{key}}

Environment variable defined in pre-request script:



Collection variable defined in pre-request script:

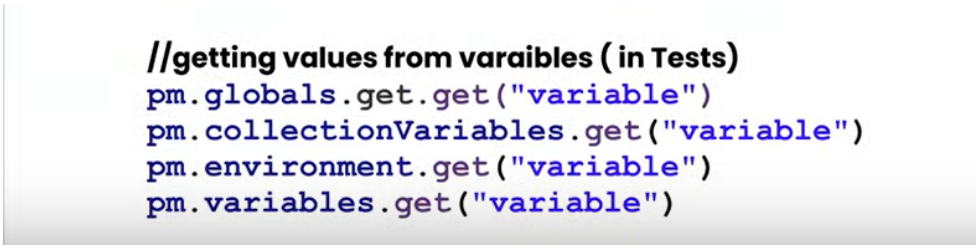


- Local variables are created during run time and also by default removed after execution and it is not stored permanently, however the global, collection and env variables are always stored permanently in their respective setup location when user creates them manually or through scripts.
- Now in order to remove these vars there are 2 ways
    - Manually remove them from their respective setup location
    - Run the scripts in post-response body

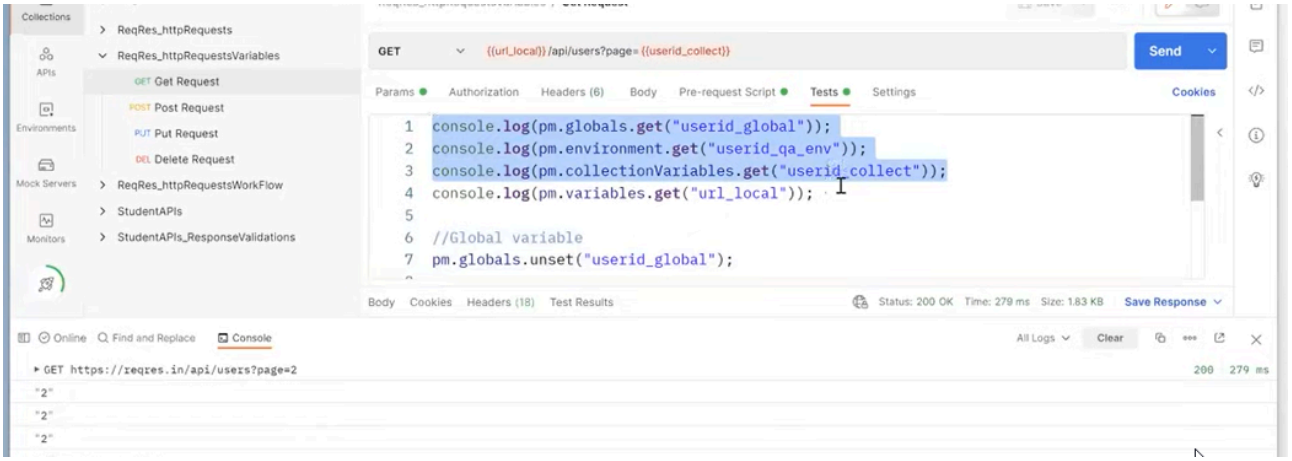
Remove Global, collection, env vars through scripts in post-response body

- We need to add below mentioned scripts in post-response body and as soon as the response is generated the scripts in post-response body will run and the intended vars wil be removed permanently

Getting values of vars through script



- **We can print any values I postman console using: console.log("variableKey");**

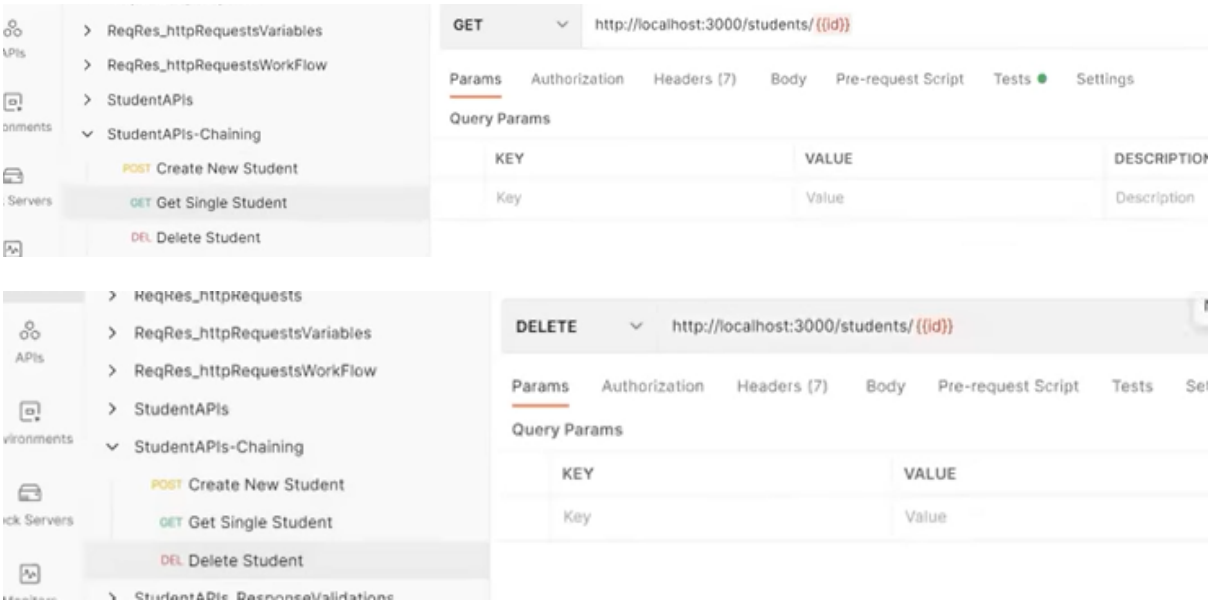


API Chaining

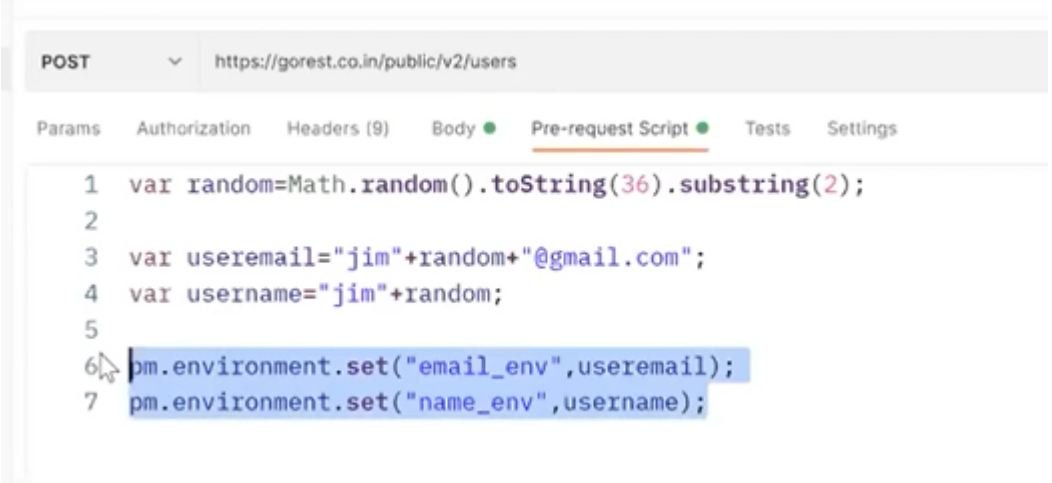
- Response generated by one API set as a request to another API is called API chaining
- Lets say we have a collection with post, get and delete requests. We can get id from post response and send it to get and delete requests:

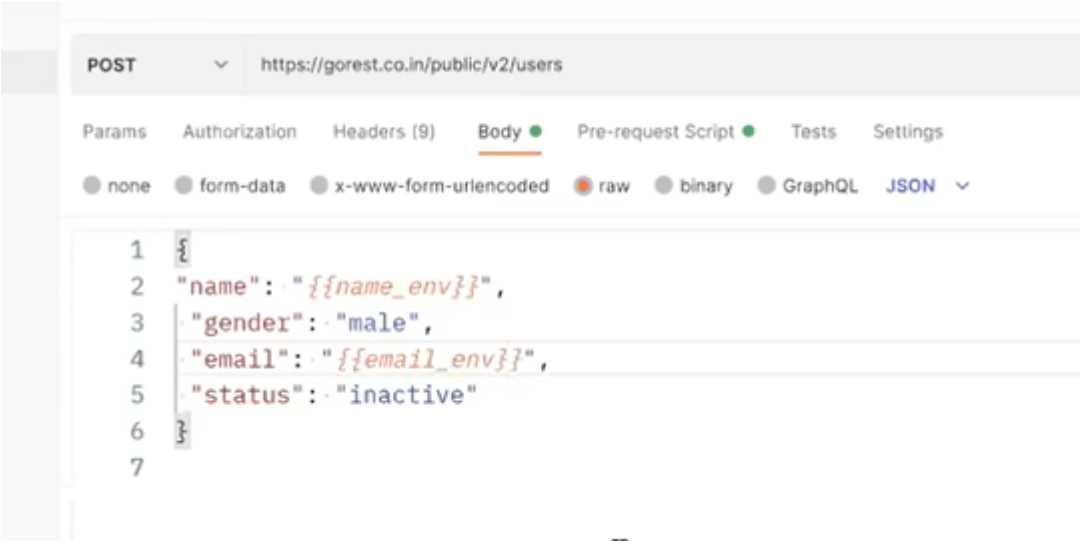
Store the response body in a var and this code will be written in post-response script section of POST request and set the ID var

Use the ID var in the url of get and delete, this will fetch the data from POST response

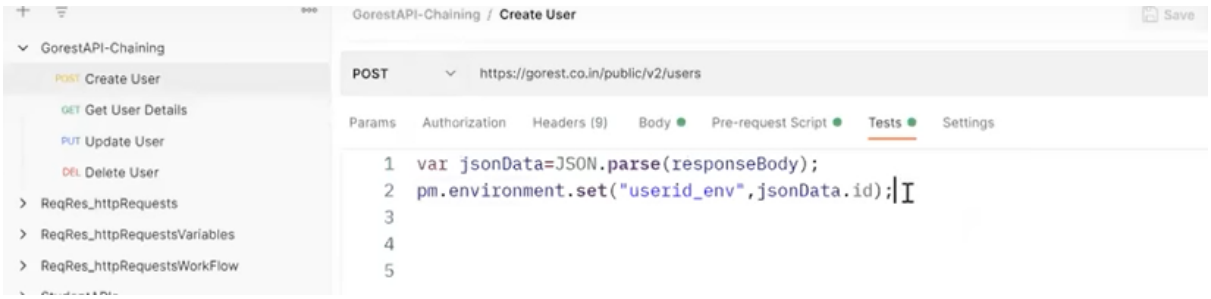


- Through Pre-request script we can generate random data such as user name and email and also send it to the request body just like using the var in request url: we add var in **“{{varName}}”**

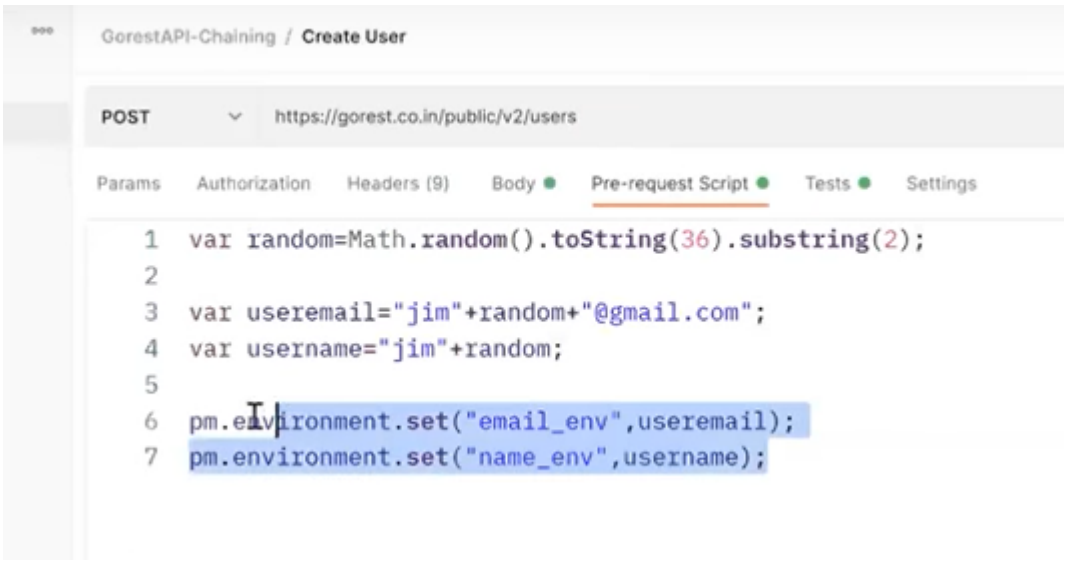


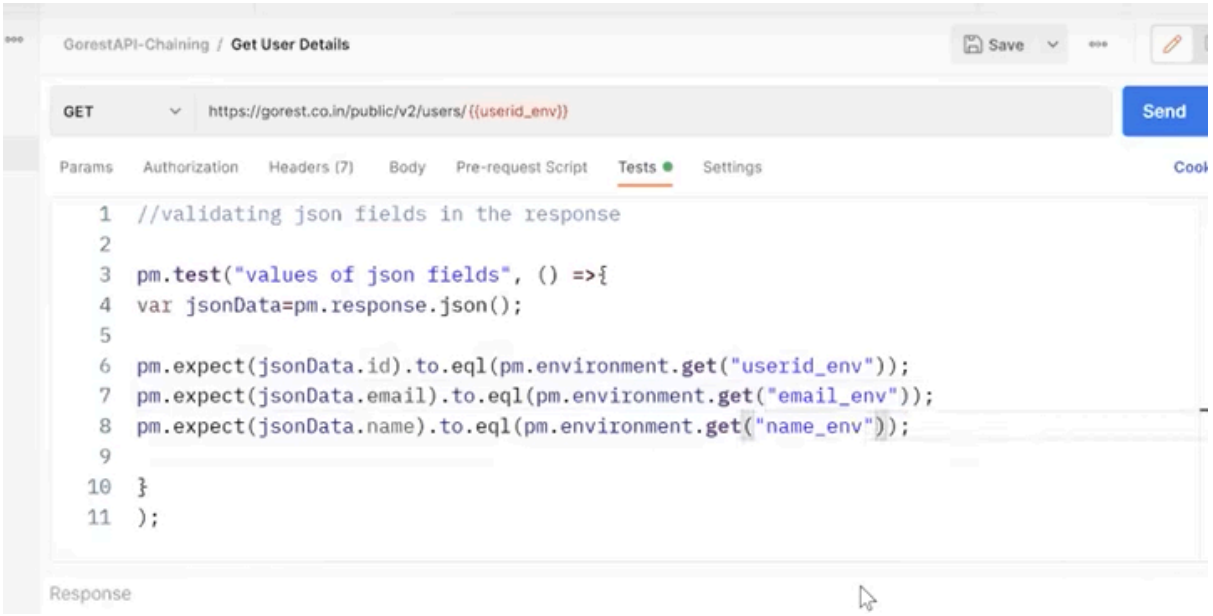


- For validations we can get the created variable from pre-request script and add it in the validation steps in post request scripts


GorestAPI-Chaining
Create User
Get User Details
Update User
Delete User
ReqRes_httpRequests
ReqRes_httpRequestsVariables
ReqRes_httpRequestsWorkFlow
GorestAPI-Chaining / Create User
POST
https://gorest.co.in/public/v2/users
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
var jsonData=JSON.parse(responseBody);
pm.environment.set("userid_env",jsonData.id);

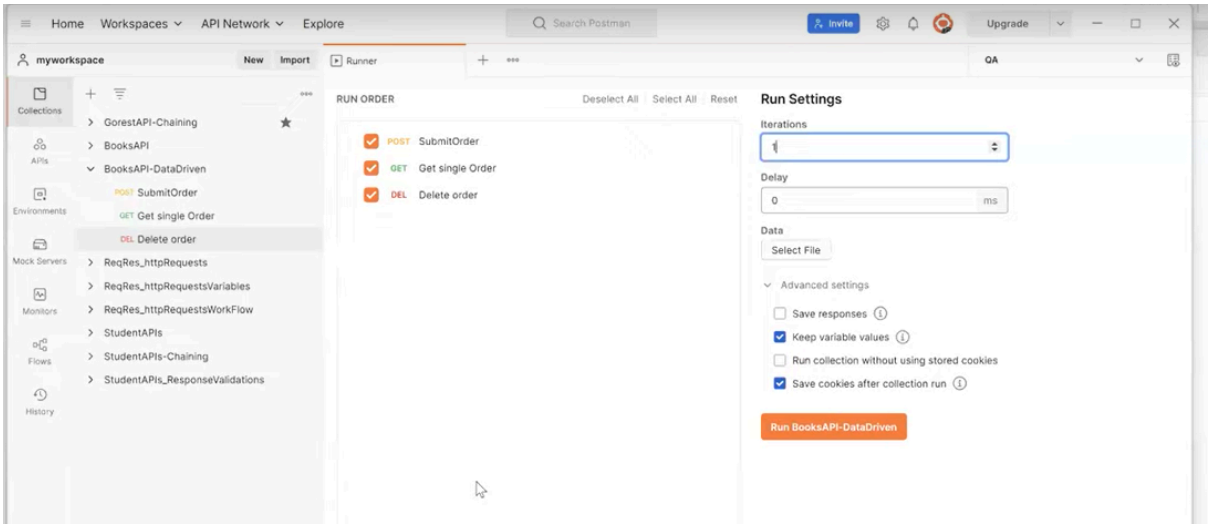

GorestAPI-Chaining / Get User Details
GET
https://gorest.co.in/public/v2/users/{{userid_env}}
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
//validating json fields in the response
pm.test("values of json fields", () =>{
var jsonData=pm.response.json();
pm.expect(jsonData.id).to.eql(pm.environment.get("userid_env"))
});


GorestAPI-Chaining / Create User
POST
https://gorest.co.in/public/v2/users
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
var random=Math.random().toString(36).substring(2);
var useremail="jim"+random+"@gmail.com";
var username="jim"+random;
pm.environment.set("email_env",useremail);
pm.environment.set("name_env",username);



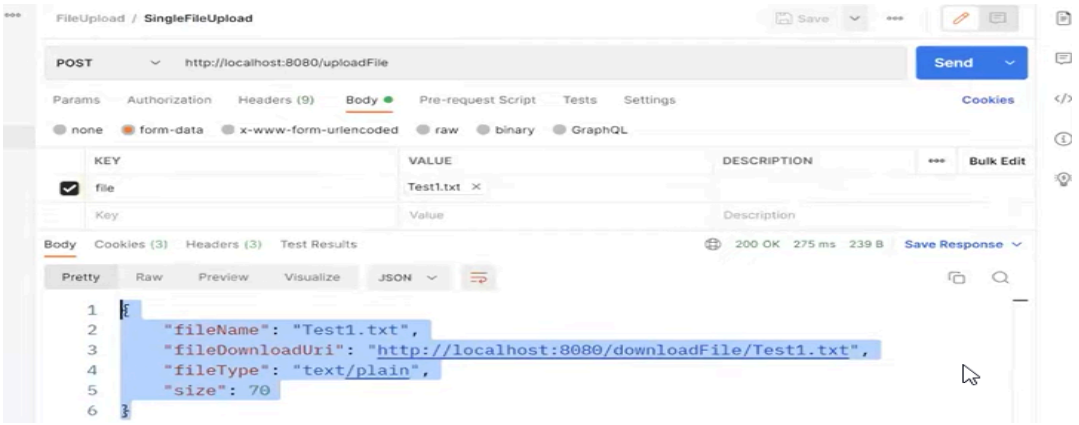
Data Driven testing

- We can store data in a json/csv/excel file and pass the key to the scripts in postman as **{{key}}**
- Right click on collection/http request and click on run and we will see a page where we can find iteration, delay and data file button under Rn settings
- Click on data file button and select the file where we store data
- Preview button will be displayed once uploaded and this will review the format of the data if the format is correct the data will be displayed in key-value pair once clicked preview button. If the format is wrong then it will display the incorrect error
- We do not need to write any specific script to get the data from external file, once uploaded the file in run page the data will be automatically fetched by the param **{{key}}** defined in the script
- Iteration field defines the number of rows of data that needs to be iterated for testing, if the file has 5 rows of data and we define iteration as 3 then the test will run 3 times and first 3 rows of data will be used for execution
- Delay field defines the rest time between each iteration



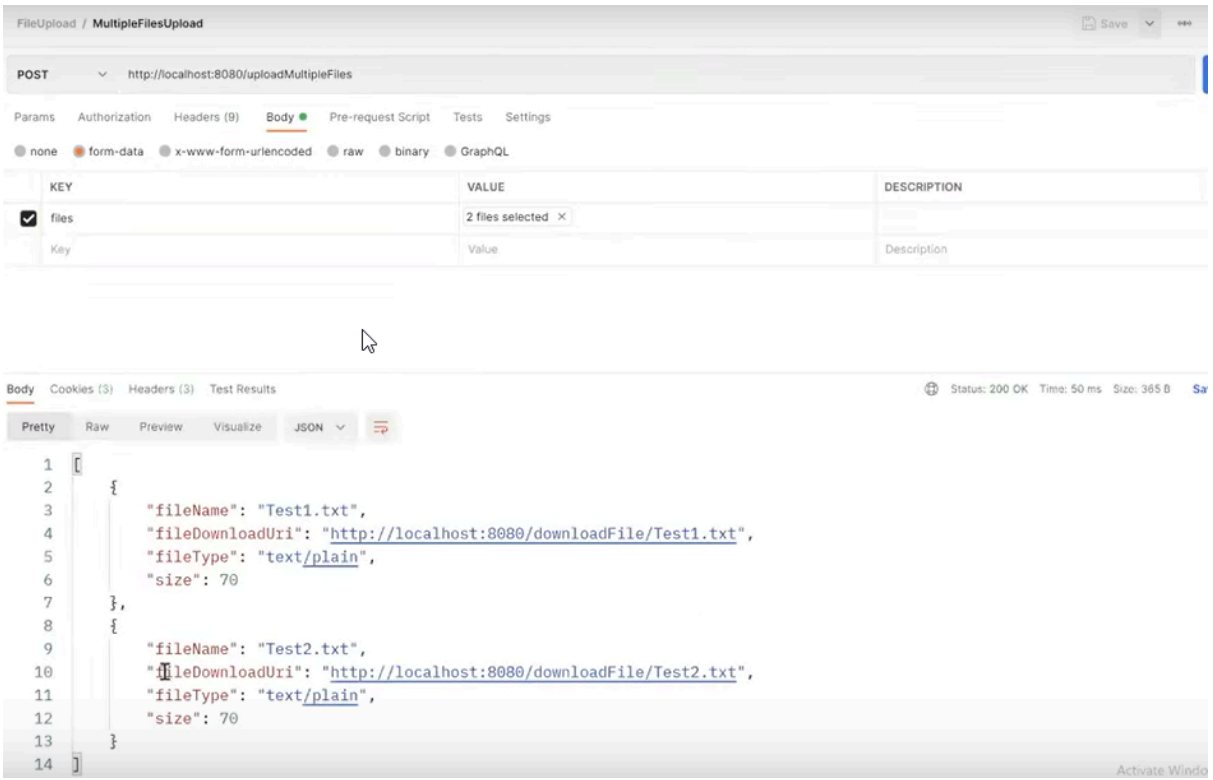
Upload File Testing

- Uploading files we will use POST request
- Click Body > **form-data** > here the key will be **file** and its type will be **file** > then in value section we will get **select file** button upon clicking it we can upload a file
- Once we send this post request we get a response something like this:



Uploading Multiple files

- Click Body > **form-data** > here the key will be **files** and its type will be **file** > then in value section we will get **select files** button upon clicking it we can select multiple files at once and upload them
- Once we send this post request, we get a response something like this:



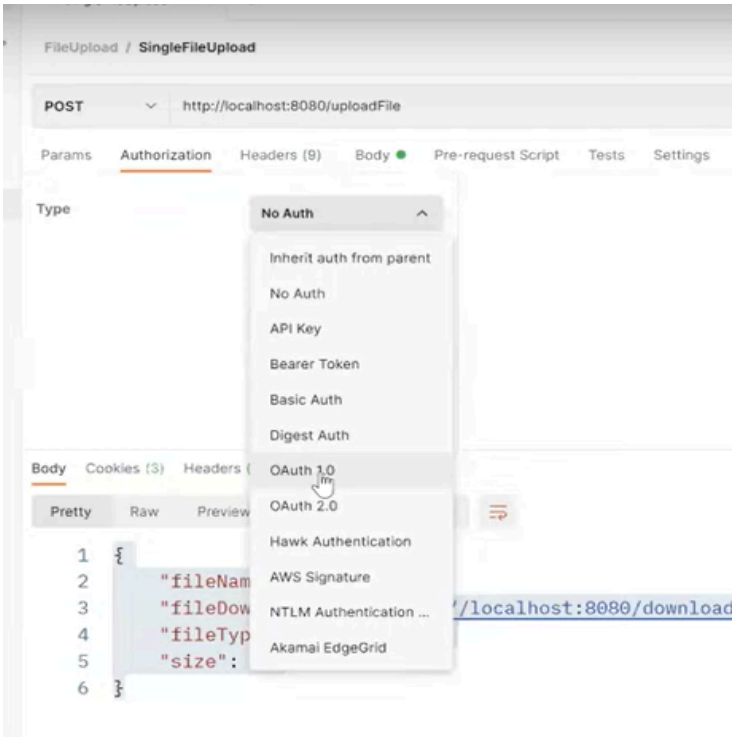
- If we select the file type as text then we need to give the path of the file in the value section.
- Only file and files keywords are used to upload a single and multiple files
- Form-data params will go as a request headers to the server

Authentications

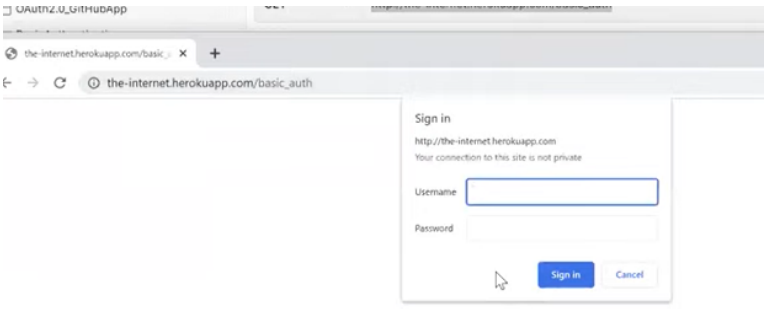
- Used to authorize and send http requests to the servers

Type

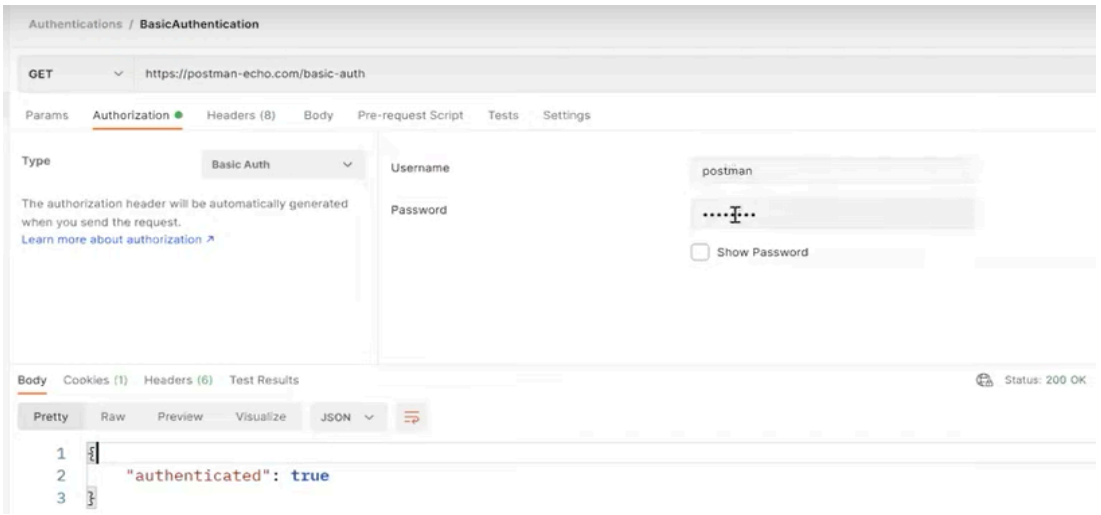
- **No Auth:** No authentication required
- **API Key:** Access key required to be generated to access API requests. It will have an API Key and value that we define in postman
- **Bearer token:** Access token required to be generated to access API requests
- **Basic Auth:** User name and password required, this is not encrypted
- **Digest Auth:** User name and password required, this is encrypted
- **OAuth 1.0:** Its process of generating authentication through 3rd party tool and its more secured than other types
- **OAuth 2.0:** Its latest version of OAuth series
- Some other newly introduced Authentications: **Hawk, AWS Signature, NTLM, Akamai EdgeGrid**



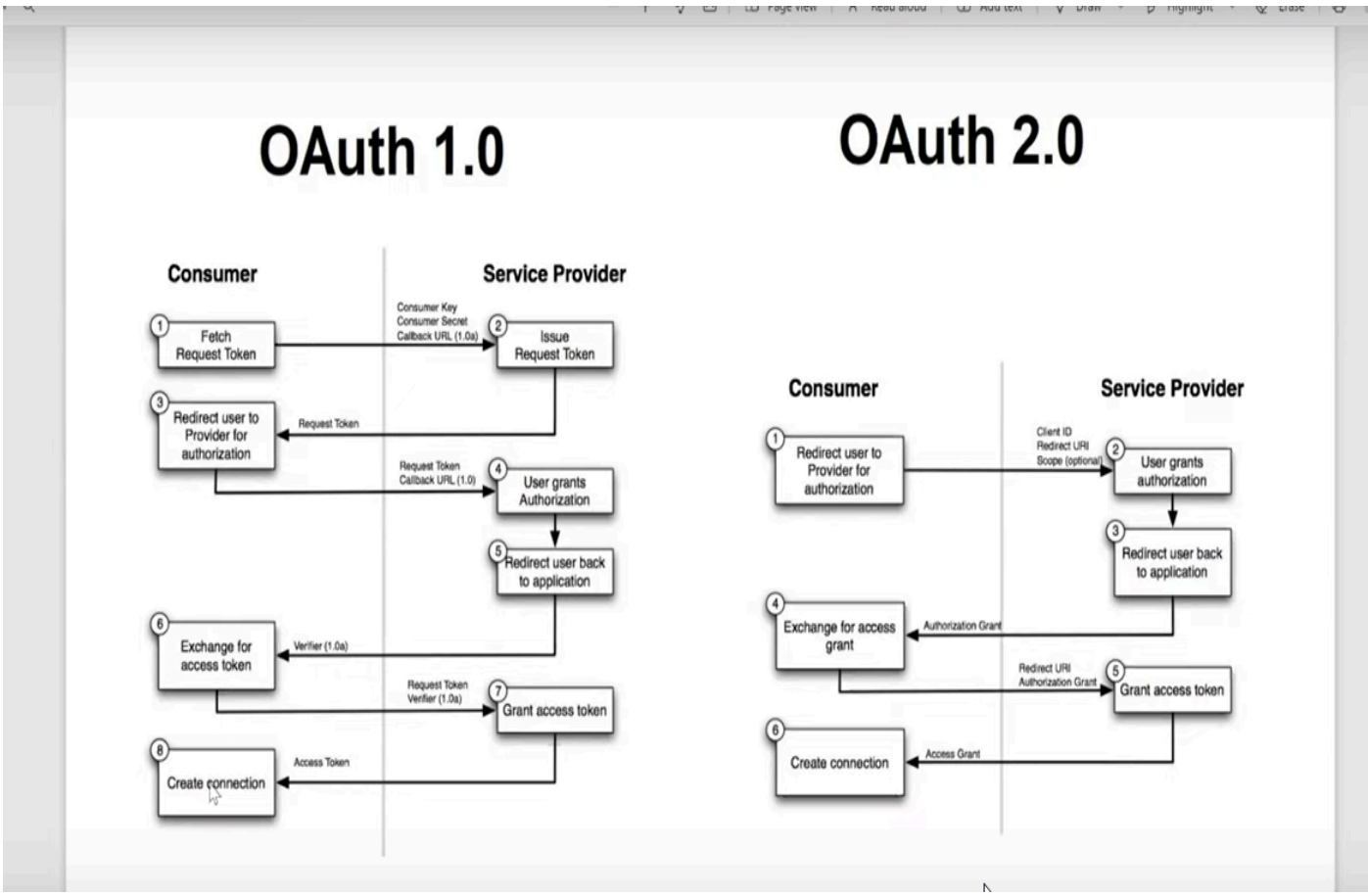
- **Basic Auth:** User name and pwd will be asked in UI just like tapestry sites



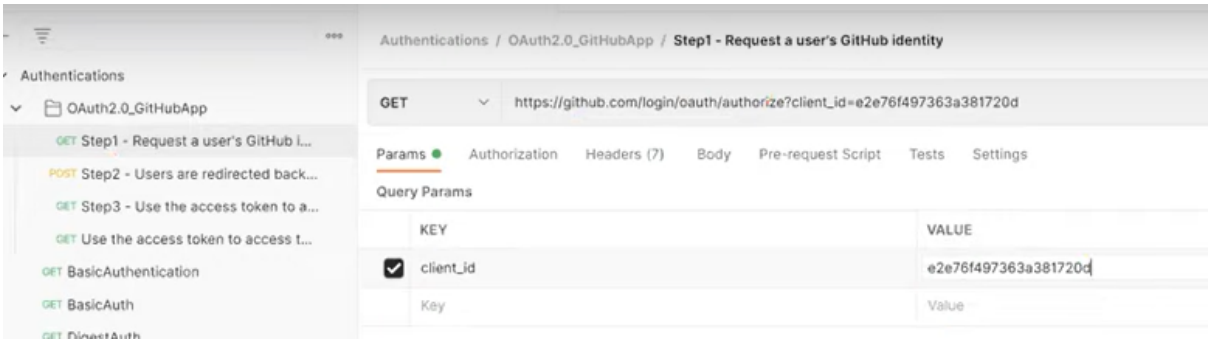
- In Postman we add creds in Authorization section



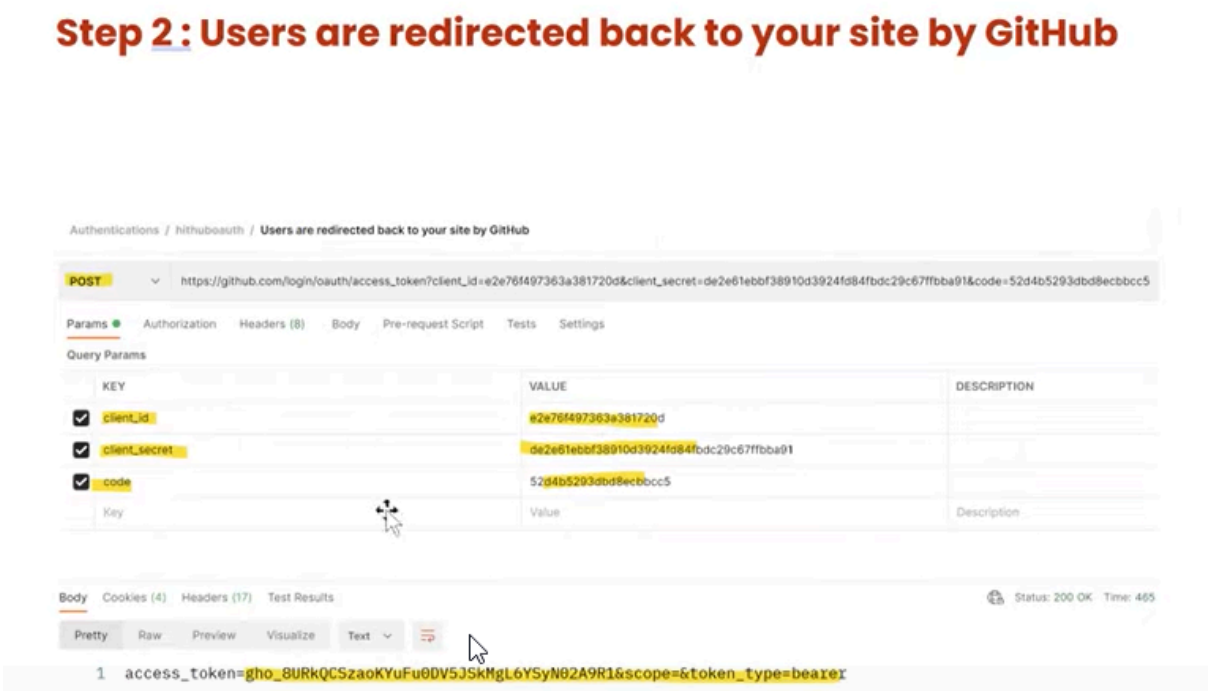
OAuth 1.0 / 2.0



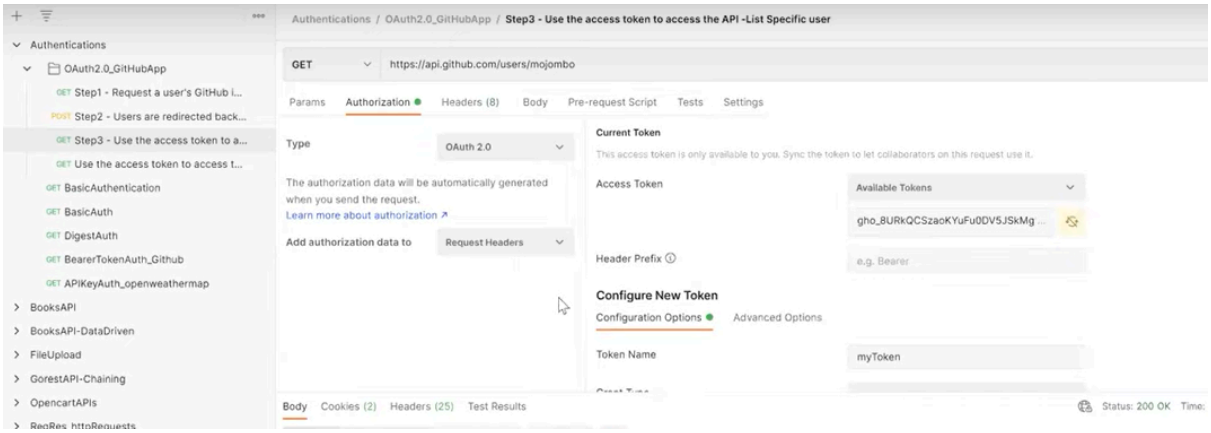
- In OAuth 1.0 an request token is generated by the server when user sends authentication request with client ID and client secret code, this token is then sent by the user to the server to get grant authorization. This extra step of getting request token is eliminated in the OAuth 2.0 version
- **Step 1:** In OAuth 2.0 Here first we need to **GET** request user's identity verification to the server by passing the url with client ID and secret code – Client ID and secret code will be created for the client/user when they have their Developer account created in the app's server



- Once we send the authorization request with client ID, the server will grant authorization to the user and a code will be generated in the response
- **Step 2:** Next step is to send a POST request to exchange the access grant by passing the client ID, secret code and the code we got from previous step



- Now we will get the OAuth access token from the server, and this access token we will define in the authorization section with OAuth selected and access token added in the access token field we now will be able to access the app's APIs
- Step 1 and 2 are done only to generate this access token and with this token we will access APIs of the app

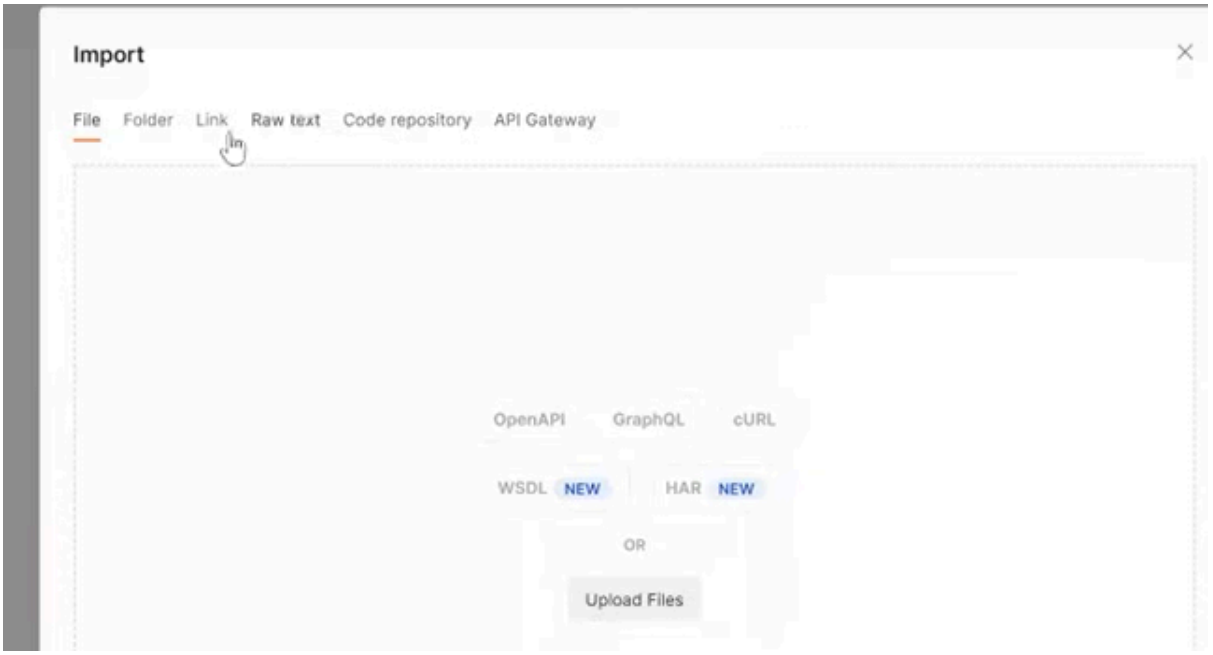


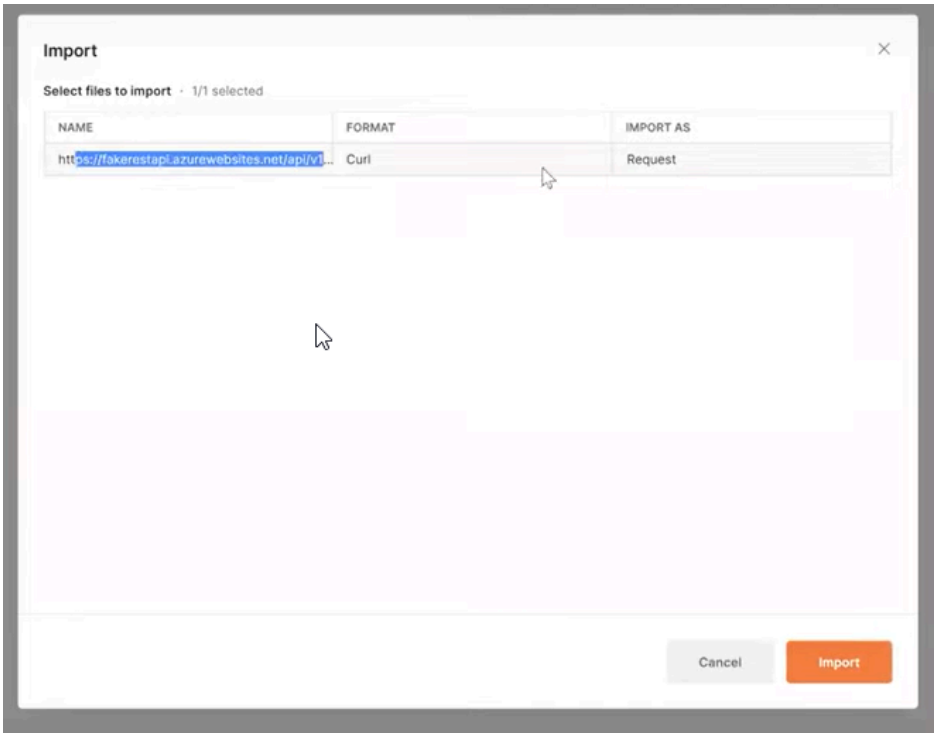
Swagger

- It's a dynamic document also called as interactive document used by developer to provide the info about an API, it will include all details such as url, request method, response data, types, etc
- Interactive docs are the types where user can perform generic interactions and the actions can connect to server and perform intended actions based on the interaction
- **Sample swagger:** https://fakerestapi.azurewebsites.net/index.html

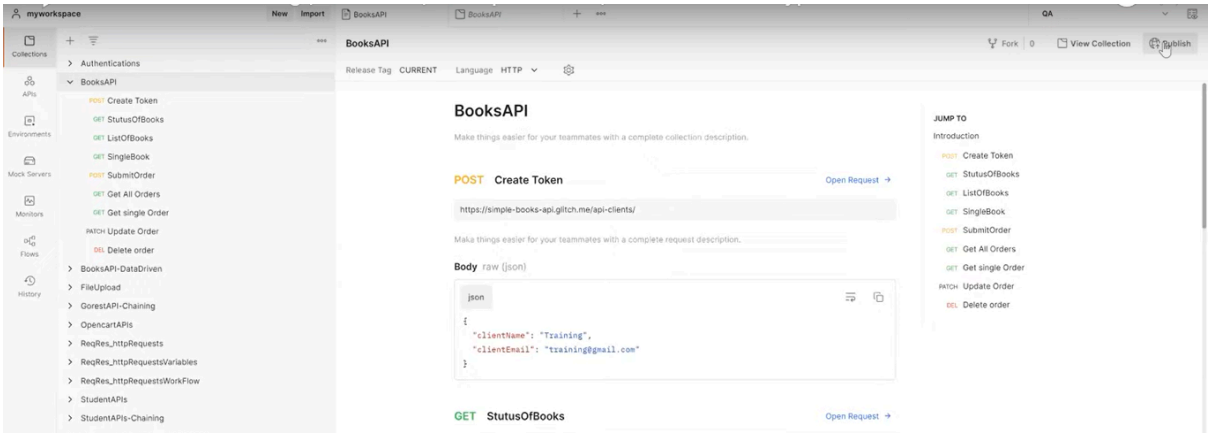
Curl: Client Url

- It’s a consolidated url with header, authorization, param, body details added
- It is used to avoid defining header, authorization, param, body details separately
- With this we can direct send an API request over the swagger, postman as well as in CLI
- We can import the curl in postman to get the request url and header/param/body added in the respective sections by > Click import > click Raw text > Paste the curl in the raw text field > click continue > click import
- Swaggers usually have curl defined
- **Sample swagger has curl: https://fakerestapi.azurewebsites.net/index.html**

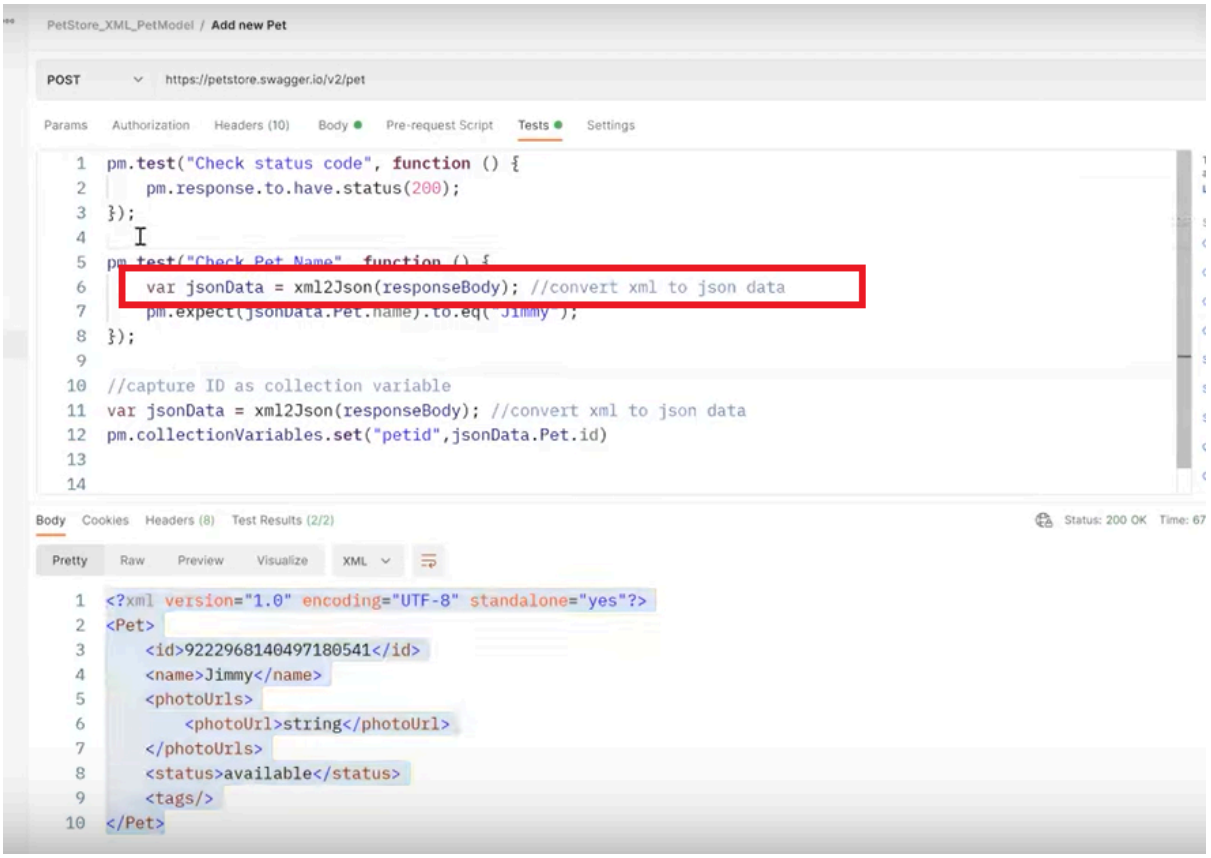




- Once we have completed testing by creating collections and its api methods we can create a document to showcase the summary of our testing from postman. Right click collection > click view documentation > and with this we can create the document of different format such as http, curl, text, etc and click publish. **Note: This will not be considered as swagger, Swagger docs are like AC and not summary.**



Note: Postman does not support v=direct validation of xml data, this needs to be converted to json and then validation can be done

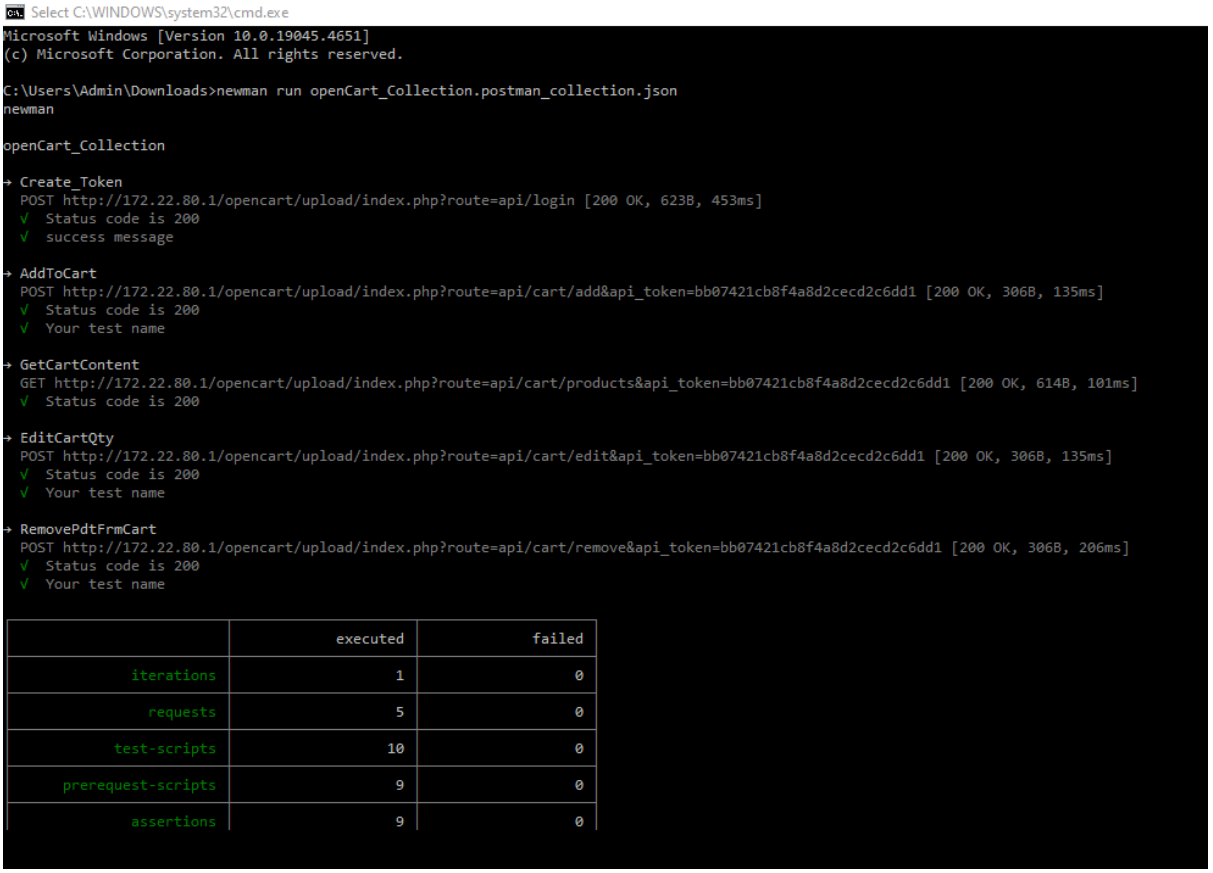


Running Collections from CLI and generating report

- It requires following to be installed:
  - Node.js
  - Npm tool of node.js (this will be pre-installed with node.js installation)
  - Newman (For execution in CLI)
  - Newman - reporter – html (for report generation)
- We need to export the collection – click on collection's 3 dots and click export
- **Note:** Newman is a library of node.js but they don't get preinstalled
- **Install Newman**
- In CLI –
  - npm install -g newman
  - npm install -g newman-reporter-html

Execute in CLI without Report generation –

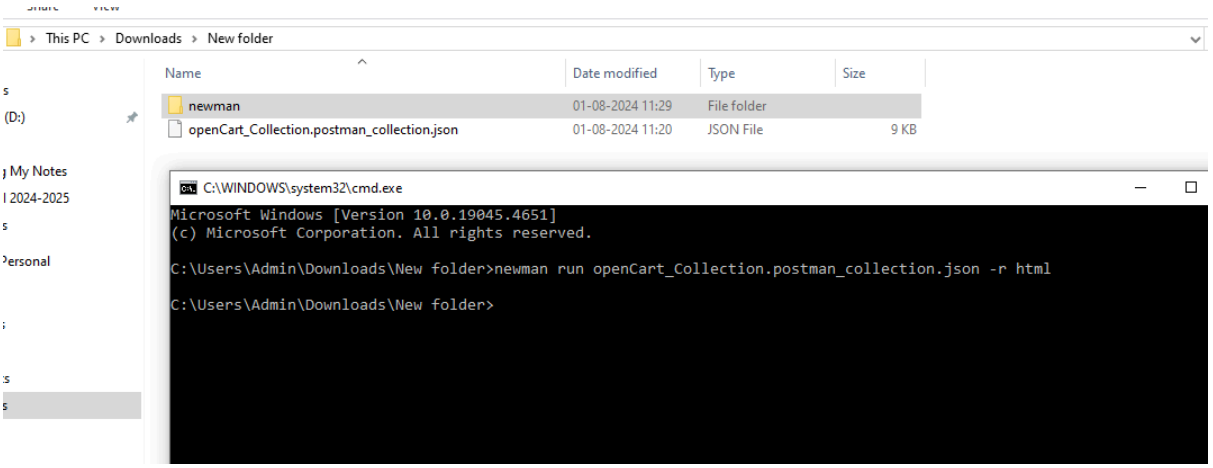
- Open CLI from collection's path
- Command – newman run <collection file name with extension>



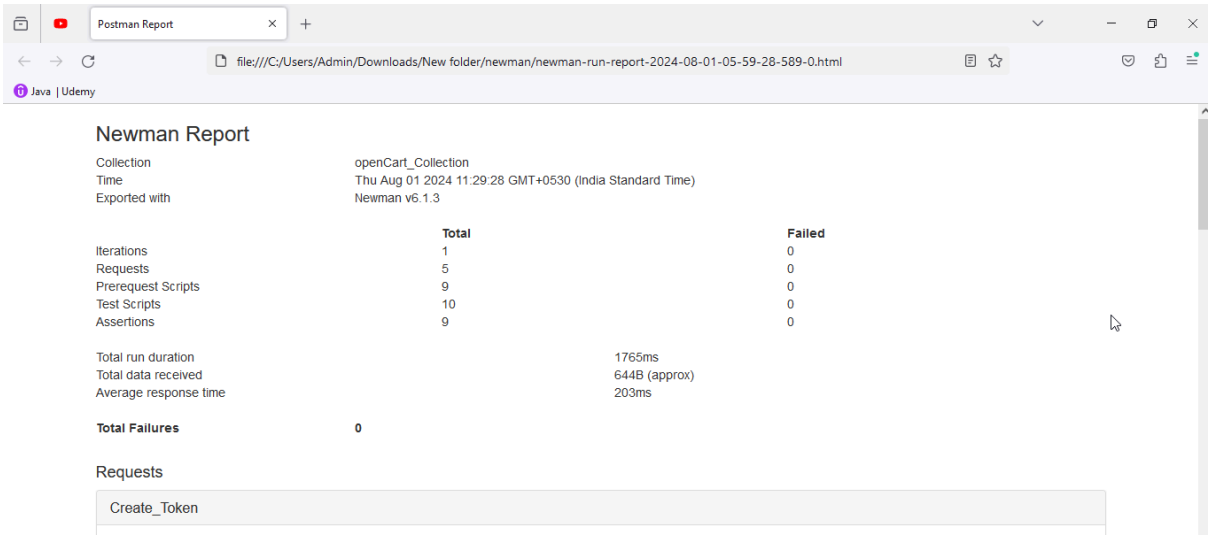
Execute in CLI with Report generation –

- Run cmd - newman run <collection file name with extension> -r html

- This wil create a folder named newman in the collection location which will have the html report



- Open html to view collection report

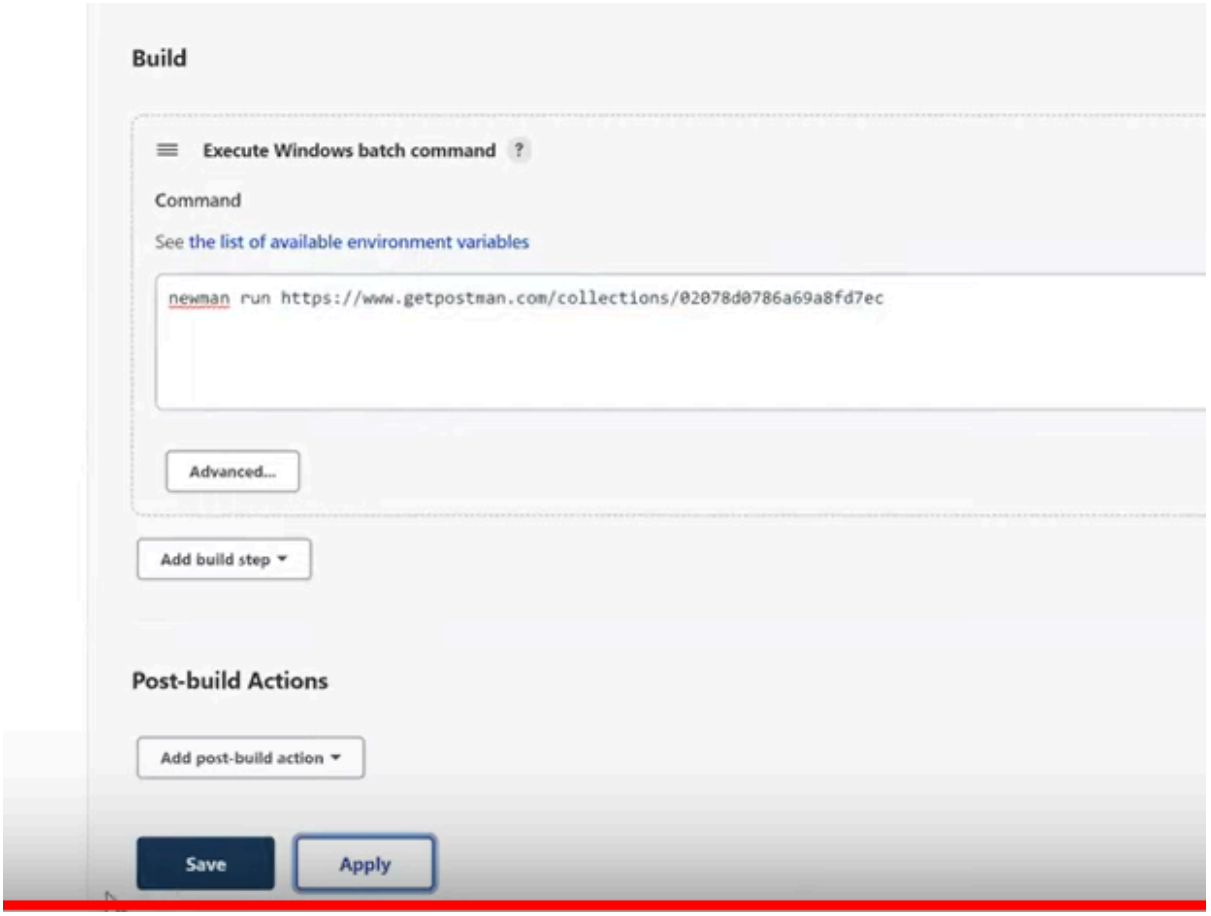


Execute Collection using it's url:

- click on collection dots > click share > select json link > copy it/share it
- Open CLI use code:
  - newman run <copied/shared link> -r html
- With this approach anyone can run the collection in their machine locally and remotely

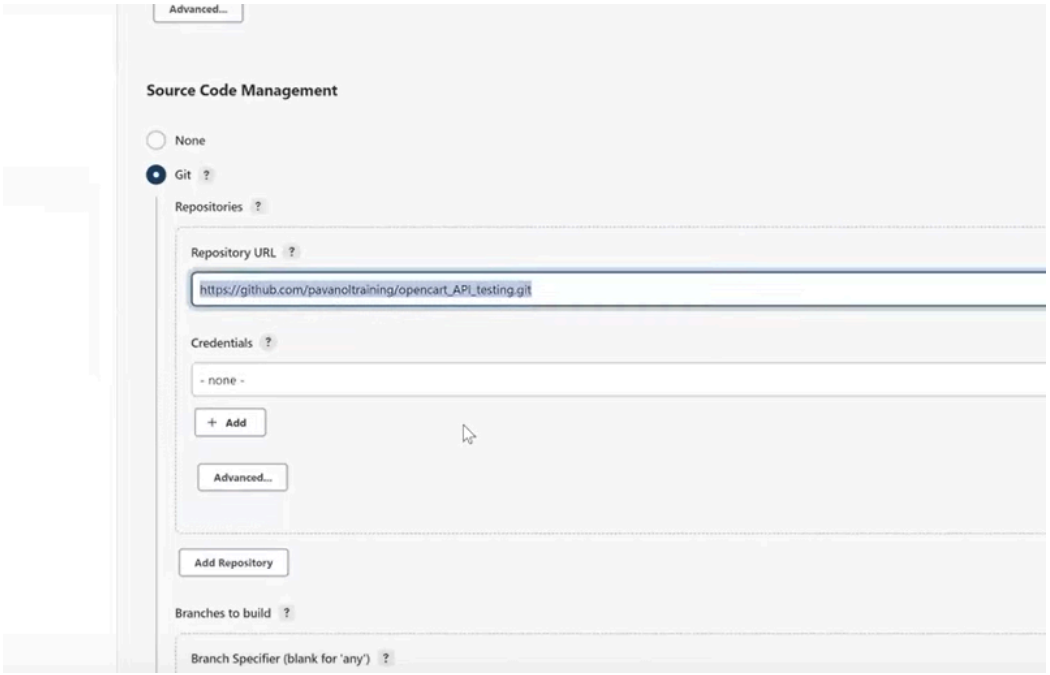
Execute Collection in Jenkins

- In Jenkins frestyle project configuration add the command with collection url in Build section and save the project, now when we run the project the windows batch command will execute the code below
  - newman run <copied/shared link> -r html



Execute Collection from Git using Jenkins

- Push file with collections json in git
- In Jenkins project configure git with it and add the newman run command with collction path in the configuration and run the project. With this we will be able to execute collection through Jenkins from git


Advanced...
Source Code Management
None
Git
Repositories
Repository URL
https://github.com/pavanoltraining/opencart_API_testing.git
Credentials
- none -
Add
Advanced...
Add Repository
Branches to build
Branch Specifier (blank for 'any')

RestAssured API Testing

- Push RestAssured is an API or library through which we can automate REST APIs
- We cant use this to test other APIs such as SOAP, GraphQL, GoogleRPC, JSON-RPC and Apache Thrift

Tech stacks required

- Eclipse
- Java 9+
- TestNG
- Maven

Maven Dependencies required

- Rest-assured
- json-path
- json
- json-schema-validator
- xml-schema-validator
- gson – It is a Java library that converts Java Objects into JSON and vice versa
- testng
- scribejava-apis – This is a 3rd party tool used to generate random test data for testing

Scripting

- 3 main keywords are used:
    - given(): Holds prerequisites – set Headers, add cookies, add query param, add auth code, set content type
    - when(): Holds requests: get put, post delete
    - then(): Holds validations: validate status code, msg, extract response/header/cookies
- We also use and() – this will hold additional validations with Then() and requests with When()
- When we start using these methods we need to add some static imports that means these imports wont be automatically get imported in the class. We can find these in rest-assured getting started page

- `io.restassured.RestAssured.*`
- `io.restassured.matcher.RestAssuredMatchers.*`
- `org.hamcrest.Matchers.*`

Note: We need to add static keyword of static imports:

- `Import static io.restassured.RestAssured.*;`

Sample.java

```java
package practice;

import org.testng.annotations.Test;

import static io.restassured.RestAssured.*;
import static io.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;

Run All
public class Sample {

    @Test
    Run | Debug
    void getUsers() {

        given()   //if we dont have any prerequisites to send in given() we can remove this method and just keep when and then

        .when()
            .get("https://reqres.in/api/users?page=2")

        .then()
            .statusCode(200)   // validate response code
            .body("page", equalTo(2)) // validate response body
            .log().all(); //print response in console
    }

}
```

Different ways to Send Json in POST request

- There are 4 different ways:
  - Using HashMap – If json is very small
  - Using org.json library
  - Using POJO Class (Plain Old Java Object)
  - Using external json file data

Using HashMap – If json is very small

```java
@Test(priority = 1)
Run | Debug
void postNewData() {

    HashMap data = new HashMap();

    HashMap address1 = new HashMap();
    address1.put("city", "India");
    HashMap addresses[] = {address1};


    data.put("firstName", "Alpha");
    data.put("lastName", "Beta");
    data.put("gender", "male");
    data.put("address", addresses);

     Response responseBody = given()
        .contentType("application/json")  // defining what content type is requested
        .body(data)

    .when()
        .post("http://localhost:3000/students")

        .then()
            .statusCode(201)
            .log().all()
            .extract().response();

     id = responseBody.jsonPath().getString("id");
```

Using org.json library – This dependency needs to be added in POM first

Non Static Imports: **org.json.JSONObject;**

Note: While passing JSON Object body it needs to be converted to string

```
void postNewData() {

    JSONObject address1 = new JSONObject();
    address1.put("city", "India");
    JSONObject addresses[] = {address1};

    JSONObject data = new JSONObject();
    data.put("firstName", "Alpha");
    data.put("lastName", "Beta");
    data.put("gender", "male");
    data.put("address", addresses);

     Response responseBody = given()
        .contentType("application/json")  // defining what content type is requested
        .body(data.toString()) //JSON Object needs to be converted to string while passing

    .when()
        .post("http://localhost:3000/students")
```

Using POJO Class (Plain Old Java Object)

- A class with getters and seters for the declared variables is called POJO class
- From this class we call setter methods in main class to set var value and call getter method to retrieve the var value

Pojo_PostRequest.java

```
public class Pojo_PostRequest {

    String firstName;
    String lastName;
    String gender;
    HashMap address[];

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public HashMap[] getAddress() {
        return address;
    }
    public void setAddress(HashMap[] address) {
        this.address = address;
    }
```

SendingJsonBodyPojoClass.java

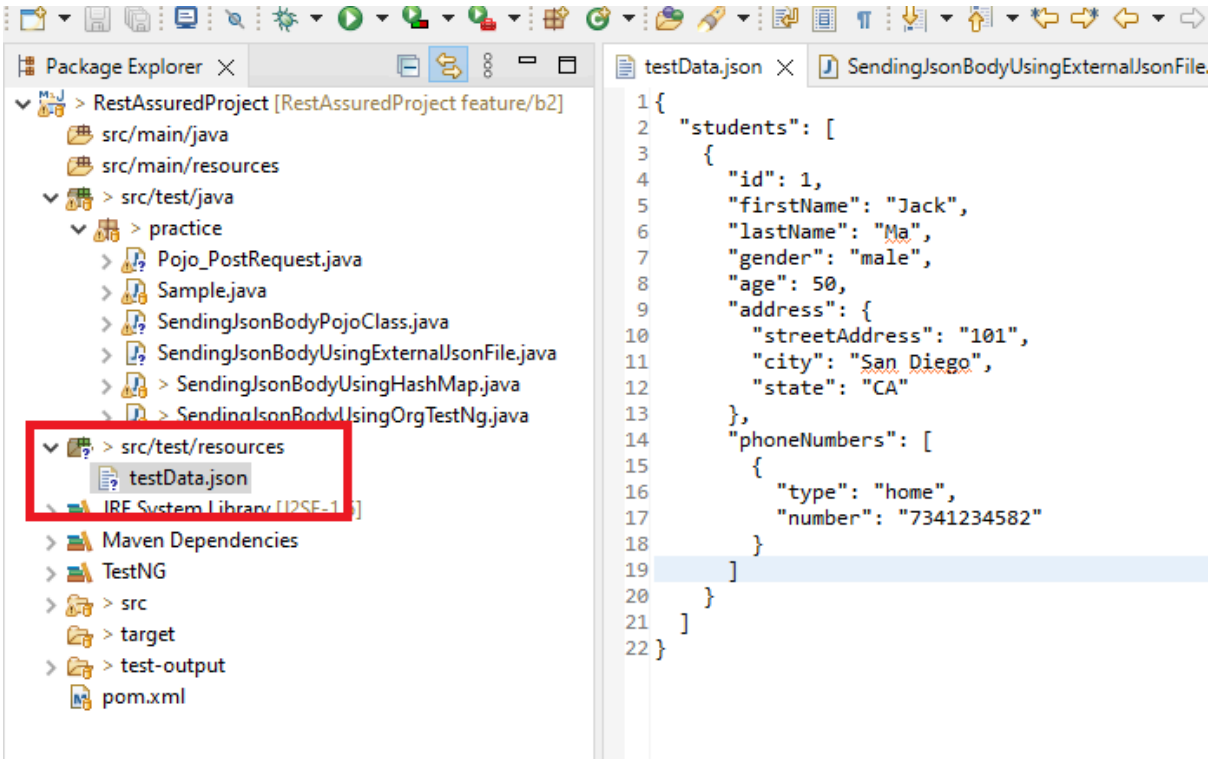
```
                .log().all();
    }

    @Test(priority = 1)
    Run | Debug
    void postNewData() {

        HashMap address1 = new HashMap();
        address1.put("city", "India");
        HashMap addresses[] = {address1};

        Pojo_PostRequest data = new Pojo_PostRequest();
        data.setFirstName("Alpha");
        data.setLastName("Beta");
        data.setGender("male");
        data.setAddress(addresses);

         Response responseBody = given()
            .contentType("application/json")  // defining what content typ
            .body(data)

        .when()
            .post("http://localhost:3000/students")

            .then()
                .statusCode(201)
                .log().all()
                .extract().response();

        id = responseBody.jsonPath().getString("id");

        System.out.println("---------------" + id);
    }
```

Using external json file data

- We need **File** object to access the file
- We need **FileReader** object to read the file

- We need **JSONTokener** object - A JSONTokener takes a source string and extracts characters and tokens from it. It is used by the JSONObject and JSONArray constructors **to parse JSON source strings**.
- We need **JsonObject** to get the data
- We add throws exception to handle file not found issue
- Imports:

```
import java.io.File;
import java.io.FileReader;
import org.json.JSONObject;
import org.json.JSONTokener;
```




```
@Test(priority = 1)
Run | Debug
void postNewData() throws FileNotFoundException {

      File file = new File("D:/eclipse22-workspace/eclipse-workspace-personal-RestAssured-project/RestAssuredProject/src/
      FileReader fr = new FileReader(file); // read the file
      JSONTokener jt = new JSONTokener(fr); // extracts source string from the file and converts the data in a token
                                            // and this is used by jsonObject to parse the source string to json
      JSONObject data = new JSONObject(jt); // access the json data


       Response responseBody = given()
          .contentType("application/json")  // defining what content type is requested
          .body(data.toString()) //JSON Object needs to be converted to string while passing

      .when()
          .post("http://localhost:3000/students")

          .then()
```

Passing Path and Query parameters

```
/**
* URL - https://reqres.in/api/users?page=2
* Domain - https://reqres.in/
* path1 - api
* path2 - users
* query param 1 - page=2
* query param 2 - id=7
*/
```

- **pathParam()** method is used to pass a single path, that is – above url path is **api/users.** Api will be path 1 and users will be path 2, we can't combine both and define it in this method
- **queryParam()** method is used to pass a single query.
    - We can't combine multiple queries
    - Query param doesn't behave like ordinary variable, it always tabs along with pathParam. We don't append them in the url after appending path variable, once path variable is defined in the url the query param will by default will be appended in sequence of query params defined. Check the image below
    - Since they are not variables and get appended by own in the url, the key should be defined exactly the same as defined in the test url

```
@Test
Run | Debug
void getUsers() {

    /**
     * URL - https://reqres.in/api/users?page=2
     * Domain - https://reqres.in/
     * path - api/users
     * query param 1 - page=2
     * query param 2 - id=7
     */

    given()
        .pathParam("path1", "api")
        .pathParam("path2", "users")
        .queryParam("page", 2)
        .queryParam("id", 5)

    .when()
        .get("https://reqres.in/{path1}/{path2}")

    .then()
        .statusCode(200)
        .log().all();
}
```

Getting and Storing Cookies

- We store the response body in the Response object and then extract cookies
- We use **getCookie** to get the single cookie data with help of cookie key info
- We use **getCookies()** method to get all cookies which will be a Map and we store it in a Map

Note: Cookie values change every time hence we can just validate if cookie is generated or not, if the cookie has certain keys, if the value matches a certain regex.

```
@Test
Run | Debug
void getUsers() {

    Response res = given()
                    .when()
                        .get("https://www.google.com/");

    // Get all cookies:
    // Cookies are in key-value pairs, hence we use HasMap as getCookies method returns a Map
    Map<String, String> getAllCookies = res.getCookies();
    System.out.println(getAllCookies);

    // get single cookie
    String cookie1 = res.getCookie("AEC");
    System.out.println(cookie1);

    // Print all cookie's keys
    System.out.println(getAllCookies.keySet());

    // Print all cookies with help of their keys
    for(String key:getAllCookies.keySet()) {
            String cookie = res.getCookie(key);
            System.out.println(key + " ===    "+ cookie);
    }
```

Getting and Validating Headers

- Many Headers data remains same such as content-type, content-encoding, server, etc and these can be validated

```
@Test
Run | Debug
void validateHeaders() {

            given()
                .when()
                .get("https://www.google.com/")
                .then()
                    .statusCode(200)
                    .header("content-encoding", "gzip")
                    .header("server", "gws");   }
@Test
Run | Debug
void getHeaders() {

            Response res = given()
                .when()
                .get("https://www.google.com/");

            // getting single header value
            String header = res.getHeader("content-encoding");
            System.out.println("single header  = "+ header);

            // getting all headers
            Headers allHeaders = res.getHeaders();  // Headers object is used to store multiple headers and no maps

            for(Header h : allHeaders) {
                System.out.println(h.getName() + " =  " + h.getValue());  }
}
// we can also use log method to log all headers:
@Test
Run | Debug
void logHeaders() {
            given()
                .when()
                .get("https://www.google.com/")
                .then()
                    .log().headers();
}
```

Parsing Json Response data for validation

Approach 1:

- Using **then()** method and its methods for validation
- This approach provides limited options for different types of validations, its difficult to perform looping, not able to get and store all response body and perform validations, etc

```
void getDataApproach1() {

    given()
        .contentType("application/json")

    .when()
        .get("http://localhost:3000/students/")

        .then()
            .statusCode(200)
            .body("students[0].gender", equalTo("Joe"));
}
```

Approach 2:

- Using TestNG Assertion library and Response object. We capture json body in an obj and then validate the data in the object
- With this we have many different options of TestNG assertions with which we can validate equals, contains, we can loop the data, generate report and many more

```
Run | Debug
void getDataApproach2() {

    Response res = given()

    .when()
        .get("https://reqres.in/api/users?page=2");

    Assert.assertEquals(res.getStatusCode(), 200);  // using TestNG Assertion library

    String email = res.jsonPath().get("data[0].email").toString(); // getting json data from res object
    Assert.assertEquals(email, "michael.lawson@reqres.in");

    Assert.assertEquals(res.jsonPath().get("data[3].first_name").toString(), "Byron");
}
```

Printing all the First Name from the response body

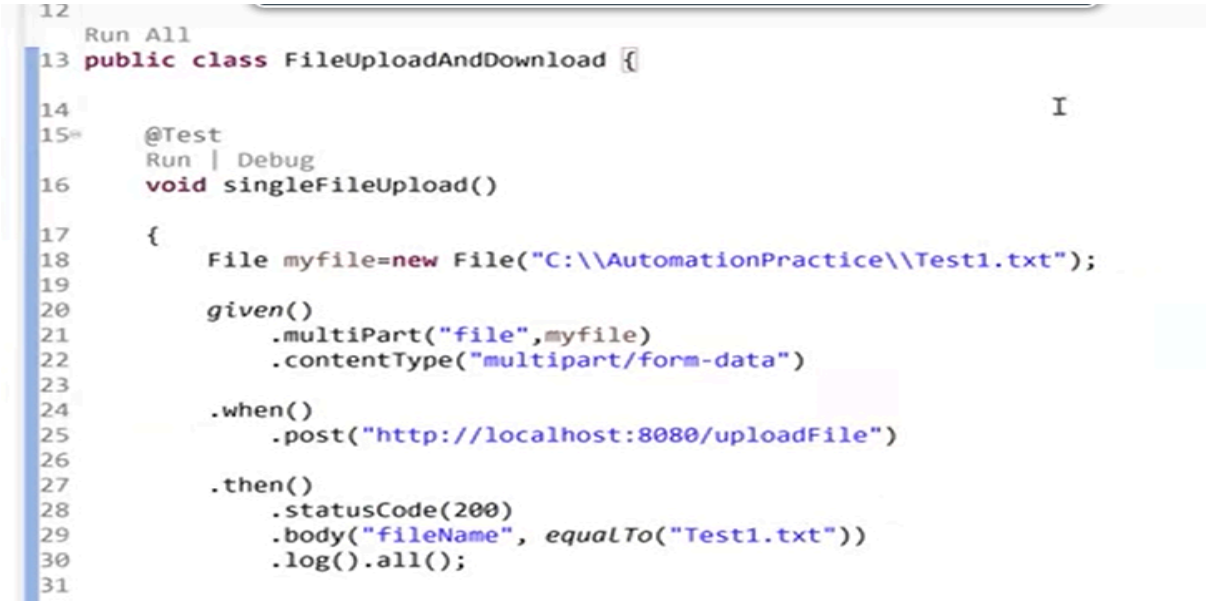
```
// Printing all the firstName from the response body
@Test
Run | Debug
void printAllFirstName() {

    Response res = given()
                    .contentType(ContentType.JSON) // this needs to be added as prerequisite so that the json body is considered as json type
                                                   // if not defined the json body is not read correctly and wil throw error
    .when()
        .get("https://reqres.in/api/users?page=2");

    Assert.assertEquals(res.getStatusCode(), 200);

    // JSONObject class from or.json library is used to convert the response to a json object
    // asString method is used from restassured library to convert the response object to string, we dont use toString method of java
    JSONObject jo = new JSONObject(res.asString());

    // getting the size of data array in the json response body
    // data is the array in the response body which has an array of objects
    int size = jo.getJSONArray("data").length();

    for(int i = 0; i < size; i++) {

        // we go to data array in the job object and then go to the object inside it at ith index and get the value of desired key
        // we convert the value to string as it will be JSONObject format
        String firstName = jo.getJSONArray("data").getJSONObject(i).get("first_name").toString();
        System.out.println(firstName);
    }
}
```

- The object in json body tends to change the positions hence, the json path approach may not work all the time as the json path uses the index position to travers to an element. In this case Approach 2 is suitable with this we can validate the data is present in the json or not and without defining positions.

Note: toString() method is used when we are converting the data of a json object and asString() method is used when we are converting entire json object into a string

File Upload Test

- Prerequisite for uploading a file via rest assured is we need to define:
    - Path of file
    - Multipart() method - Multipart Upload allows a single object to be uploaded as a collection of parts rather than as one single part
    - Content type



```
Run All
public class FileUploadAndDownload {

    @Test
    Run | Debug
    void singleFileUpload()

    {
        File myfile=new File("C:\\AutomationPractice\\Test1.txt");

        given()
            .multiPart("file",myfile)
            .contentType("multipart/form-data")

        .when()
            .post("http://localhost:8080/uploadFile")

        .then()
            .statusCode(200)
            .body("fileName", equalTo("Test1.txt"))
            .log().all();

```

Uploading multiple files

```
@Test
Run | Debug
void multipleFilesUpload()

{
     File myfile1=new File("C:\\AutomationPractice\\Test1.txt");
     File myfile2=new File("C:\\AutomationPractice\\Test2.txt");

     given()
          .multiPart("files",myfile1)
          .multiPart("files",myfile2)
          .contentType("multipart/form-data")

     .when()
          .post("http://localhost:8080/uploadMultipleFiles")

     .then()
          .statusCode(200)
          .body("[0].fileName", equalTo("Test1.txt"))
          .body("[1].fileName", equalTo("Test2.txt"))

          .log().all();
```

Download file

```
@Test(priority=2)
Run | Debug
void fileDownload()

{
     given()

     .when()
          .get("http://localhost:8080/downloadFile/Test1.txt")
     .then()
          .statusCode(200)
          .log().body();
     }
```

Json Schema validation

- Its about validating the json data type validation whereas response validation is validating the data value


```
import static io.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;

//json ---> jsonschema converter
//  https://jsonformatter.org/json-to-jsonschema


Run All
public class JSONSchemaValidation {

    @Test
    Run | Debug
    void jsonschemavalidation()

    {
        given()

        .when()
            .get("http://localhost:3000/store")
        .then()
            .assertThat().body(JsonSchemaValidator.matchesJsonSchemaInClasspath("storeJsonSchema.json"));
    }
```

Note: Postman doesn't support xml schema validation but RestAssured supports xml schema validation

Serialisation and De-serilisation

- Converting POJO/java object to json and passing this as the request to the server/DB/file is **Serialisation**
- Converting json to POJO/ java object and retrieving it from server/DB/file along with response is **De-serialisation**
- When we send the request body in script either via hashmap/pojo internally rest-assured follows serialisation process
- We can also manually.explicity do a serialisation by using ObjectMapper class from Jackson API – import com.fasterxml.jackson.databind.ObjectMapper

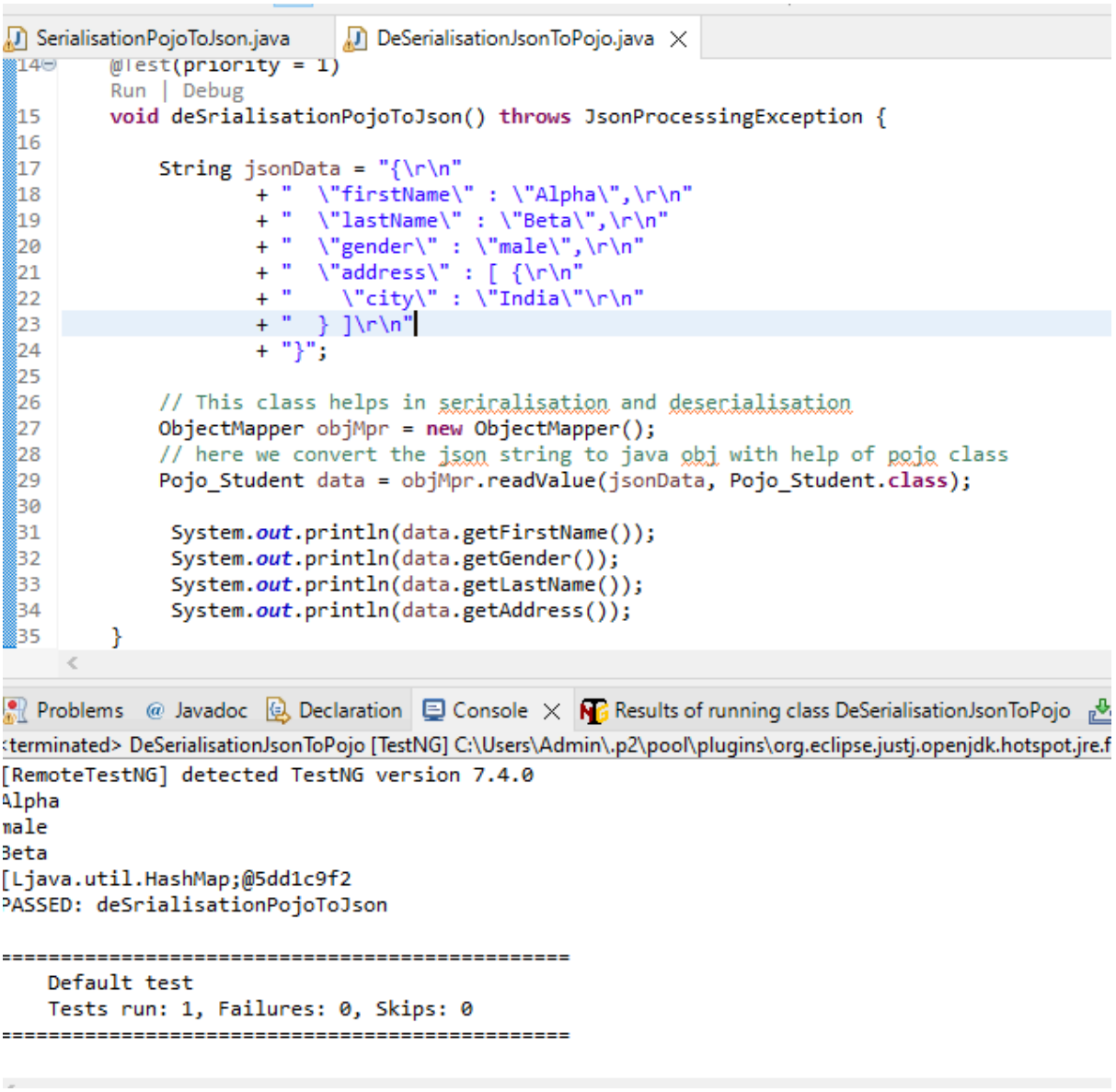
SerialisationPojoToJson.java

```
7  import com.fasterxml.jackson.core.JsonProcessingException;
8  import com.fasterxml.jackson.databind.ObjectMapper; //import for ObjectMapper
9
   Run All
10 public class SerialisationPojoToJson {
11
12
13
14     @Test(priority = 1)
       Run | Debug
15     void serialisationPojoToJson() throws JsonProcessingException {
16
17         HashMap address1 = new HashMap();
18         address1.put("city", "India");
19         HashMap addresses[] = {address1};
20
21         Pojo_PostRequest data = new Pojo_PostRequest();
22         data.setFirstName("Alpha");
23         data.setLastName("Beta");
24         data.setGender("male");
25         data.setAddress(addresses);
26
27         // This class converts the java obj to json obj
28         ObjectMapper objMpr = new ObjectMapper();
29         // here we convert the java obj into json but in string format
30         String jsonData = objMpr.writerWithDefaultPrettyPrinter().writeValueAsString(data);
31
32          System.out.println(jsonData);
33     }
```

Problems | Javadoc | Declaration | Console | Results of running class SerialisationPojoToJson | Git Staging

```
<terminated> SerialisationPojoToJson [TestNG] C:\Users\Admin\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86
[RemoteTestNG] detected TestNG version 7.4.0
{
  "firstName" : "Alpha",
  "lastName" : "Beta",
  "gender" : "male",
```

- De-serialisation is also done with ObjectMapper class and we can use pojo class to read the data



Authorizations

- Authentication: It is validation of user and/or use credentials credentials, whether the user detail is authentic
- Authorization: It is the validation of permission/access to the valid user, whether the user is authorized
- Types of Authentications/Authorizations supported by RestAssured
    - Basic
    - Digest
    - Preemptive
    - Bearer token
    - Oauth 1.0, 2.0
    - API Key
- Basic auth, Digest and Preemptive authentications are same they require username and pwd for authentication, only difference is the internal algorithm used for execution

- Authentications of RestAssured has different mechanism in comparison to Postman authentications
- Dev team decides the type of authentication to be used for APIs

Basic

```
Run All
public class Authentications {


    @Test(priority=1)
    Run | Debug
    void testBasicAuthentication()

    {
        given()
            .auth().basic("postman","password")
        .when()
            .get("https://postman-echo.com/basic-auth")

        .then()
            .statusCode(200)
            .body("authenticated",equalTo(true))
            .log().all();

    }
```

Digest

```
@Test(priority=2)
Run | Debug
void testDigestcAuthentication()

{
    given()
        .auth().digest("postman","password")
    .when()
        .get("https://postman-echo.com/basic-auth")

    .then()
        .statusCode(200)
        .body("authenticated",equalTo(true))
        .log().all();

}
```

Preemptive

```
@Test(priority=3)
Run | Debug
void testPreemptivecAuthentication()

{
    given()
        .auth().preemptive().basic("postman","password")
    .when()
        .get("https://postman-echo.com/basic-auth")

    .then()
        .statusCode(200)
        .body("authenticated",equalTo(true))
        .log().all();
```

Bearer Token

```
@Test(priority=4)
Run | Debug
void testBearerTokenAuthentication()

{
    String bearerToken="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";

    given()
        .headers("Authorization","Bearer "+bearerToken)

    .when()
        .get("https://api.github.com/user/repos")

    .then()
        .statusCode(200)
        .log().all();
```

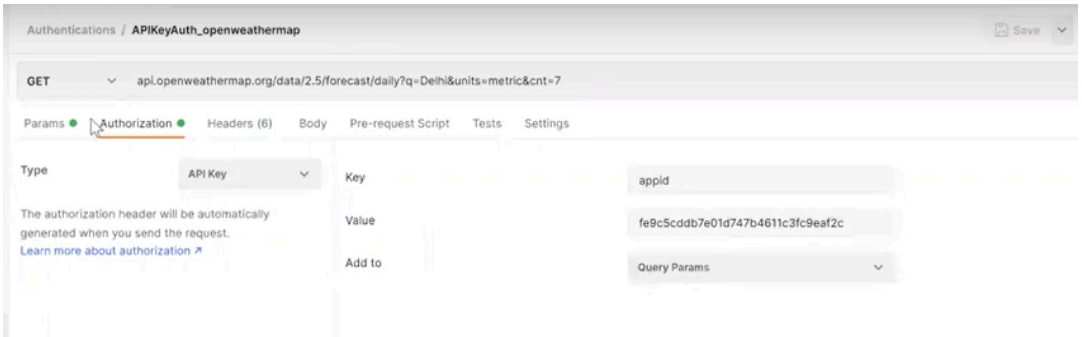
OAuth1: It needs 4 data mentioned in screenshot

```
@Test
Run | Debug
void testOAuth1Authentication()

{
    given()
        .auth().oauth("consumerKey","consumerSecrat","accessToken","tokenSecrate")
    .when()
        .get("url")
    .then()
        .statusCode(200)
        .log().all();

}
```

OAuth2: Only Access token is required. This is widely used by dev team

```
@Test
Run | Debug
void testOAuth2Authentication()

{
    given()
    .auth().oauth2("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")
.when()
    .get("https://api.github.com/user/repos")
.then()
    .statusCode(200)
    .log().all();
}
```

API Key: They can added as query param or as header it is decided by dev team



Approach 1

```
@Test
Run | Debug
void testAPIKeyAuthentication()

{
    //Method1
    given()
        .queryParam("appid","fe9c5cddb7e01d747b4611c3fc9eaf2c") //appid is APIKey
    .when()
        .get("https://api.openweathermap.org/data/2.5/forecast/daily?q=Delhi&units=metric&cnt=7")
    .then()
        .statusCode(200)
        .log().all();

}
```

Approach 2 – query param and path param defined separately,

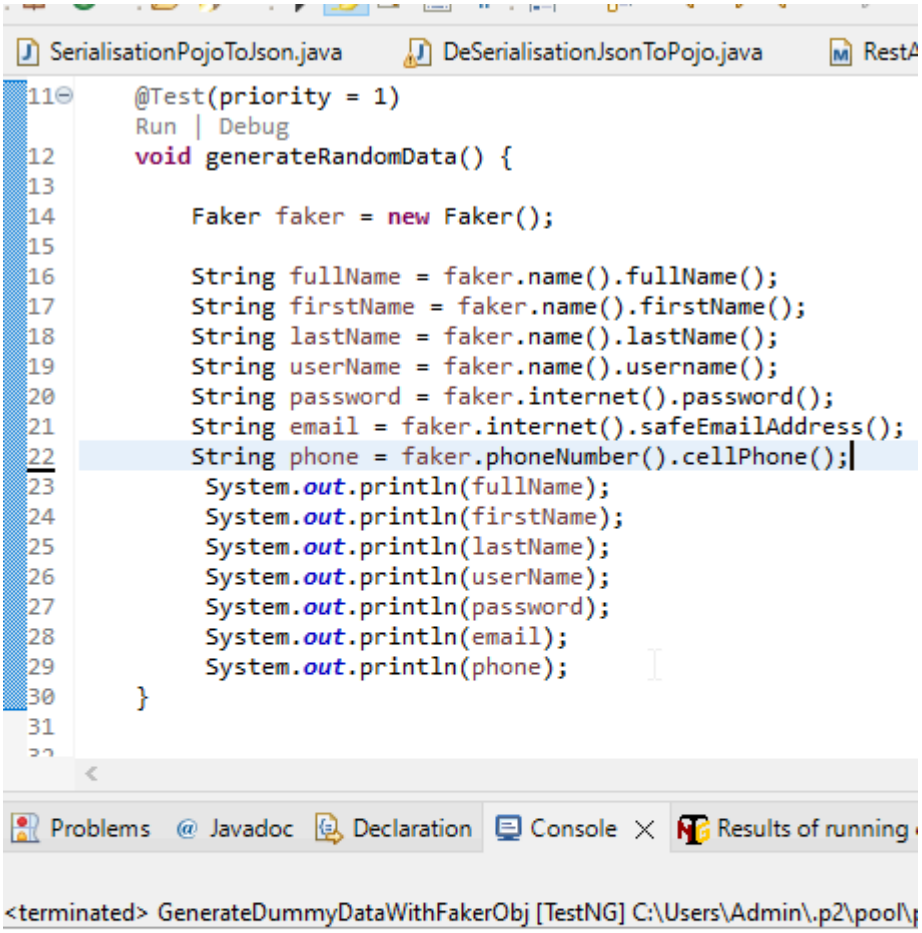
Note – No need to define query param key in url as it will be taken in Backend but path param we need to define

```
//Method2

given()
     .queryParam("appid","fe9c5cddb7e01d747b4611c3fc9eaf2c")

     .pathParam("mypath","data/2.5/forecast/daily")

     .queryParam("q", "Delhi")

     .queryParam("units", "metric")

     .queryParam("cnt", "7")


.when()
     .get("https://api.openweathermap.org/{mypath}")

.then()
     .statusCode(200)
     .log().all();
```

Faker Library

- Its is a java library used to generate random dummy data for testing during runtime
- We add a dependency in POM

```
<dependency>
  <groupId>com.github.javafaker</groupId>
  <artifactId>javafaker</artifactId>
  <version>1.0.2</version>
</dependency>
```



Cases to travers json response:

1. **Json response which starts as object and has arrays**

{

[

“key”:”value”

]

}

JSONObject jo = new JSONObject(response.asString);

jo.getJSONArray(0) .get(“key”);

2. **Json response starts as json array and has json objects**

[

{

“key”:”value”

}

]

JSONArray ja = new JSONArray(response.asString);

ja. getJSONObject(0).get(“key”);

3. **Json response starts with array and has object and this object has an array**

[

{

[

“key”:”value”

]

}

]

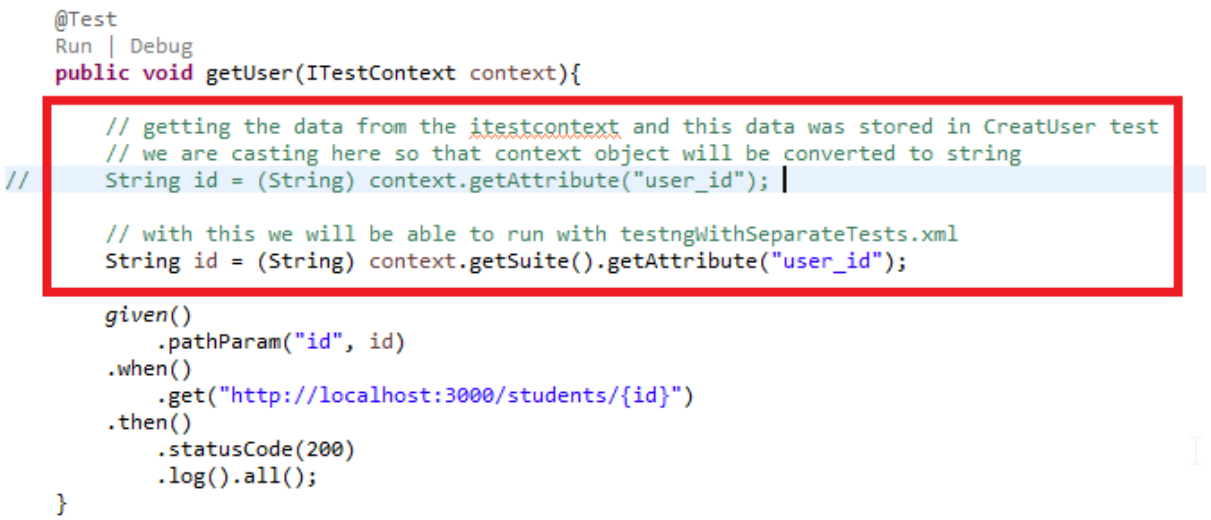
JSONArray ja = new JSONArray(response.asString);

Ja. getJSONObject(0).getJSONArray(0) .get(“key”);

Chaining

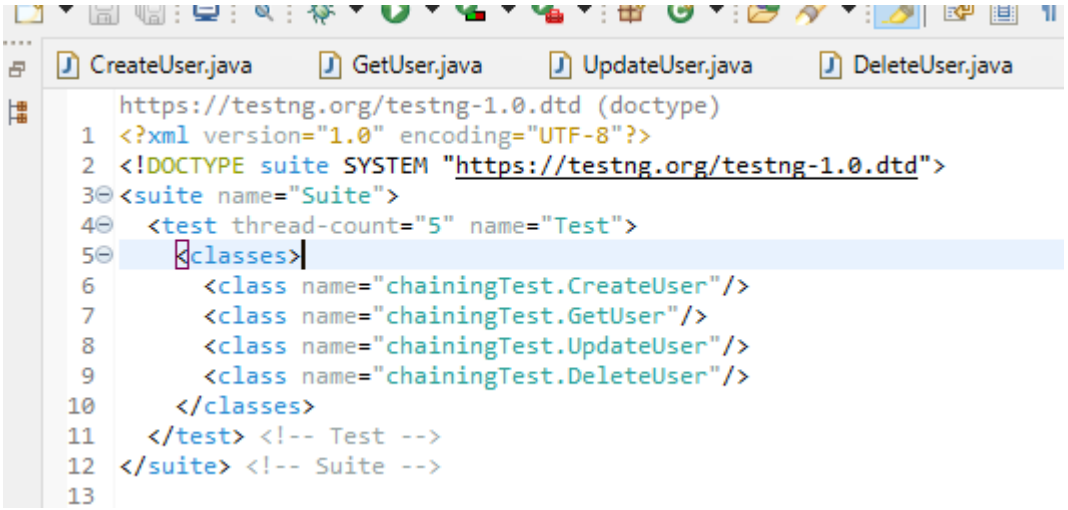
- We can use ITestConext Testng listener to set the data in one class and get the data in other classes
- We can set the context and this data will become the global variable
- If we use getSuite method to set attribute then the var will be available within entire suite
- If we don't use getSuite method to set attribute then var will be available within test level

- We can use getContext method in other class to call this global variable
- If we use getSuite method to set attribute then the var will be available within entire suite



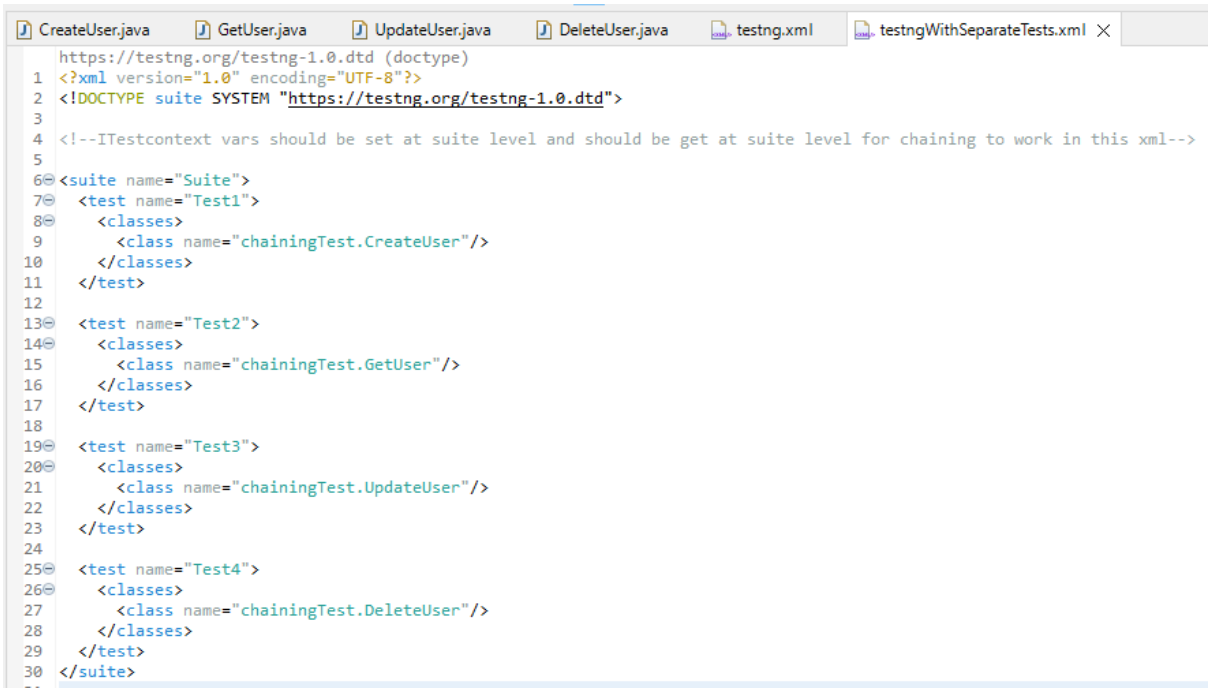
- To run these chained classes we need to use xml file to run all the chained classes together in sequence/parallel. We cannot ru them individually outside xml ad these classes are linked together

TestNG xml to run the classes in single test



TestNG xml to run the classes in different test

- If the itestcontext vars are not set with getSuite method then the chaining wil not work
- Chaining will work only when vars are set at suite level
- ITestcontext vars should be set at suite level and should be get at suite level for chaining to work in this xml



- ITestcontext vars set at suite level and get at suite level will also work for chaining with single test xml as data at suite level will be accessible by all elements within suite

API Framework Development

- **Framework** is created to maintain all project related files
- **Objective:**
    - Readability
    - Reusability
    - Maintainability

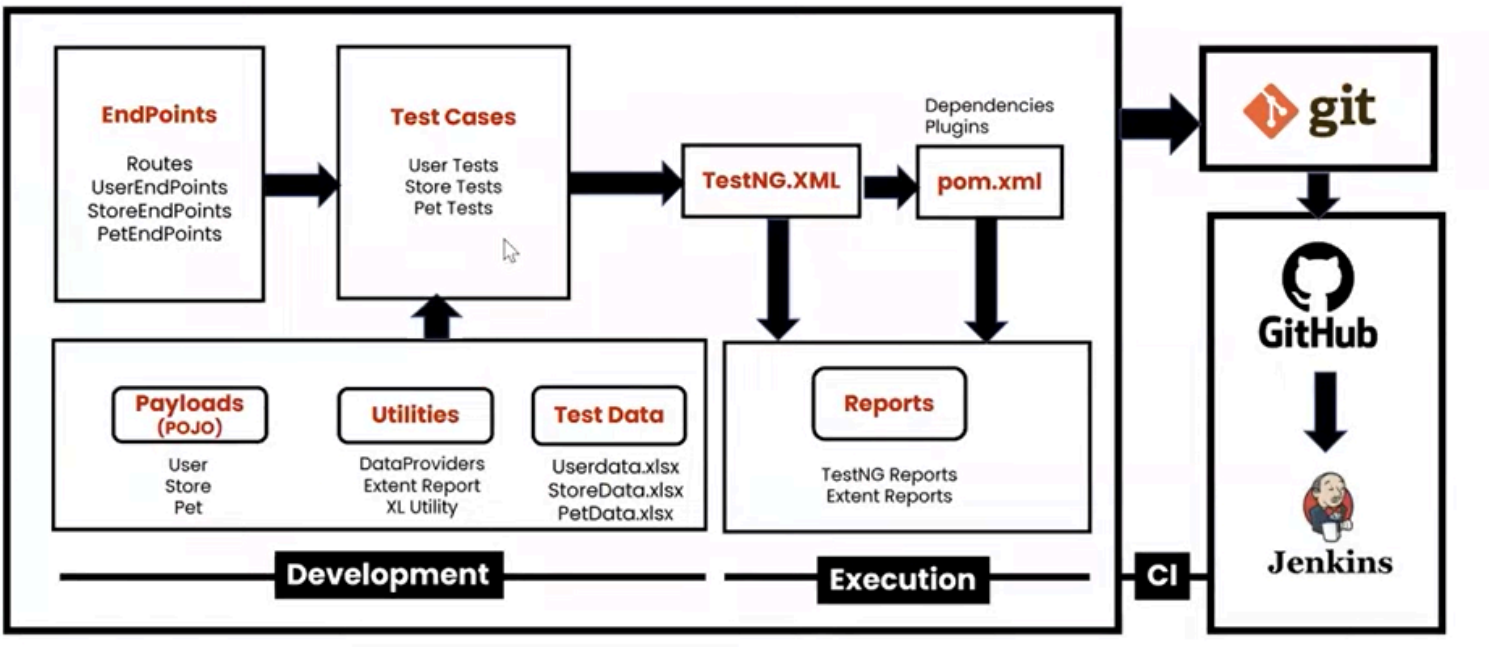
Phases of Framework creation

1. Understanding the requirement:
    a. We user Static docs such as Functional specifications
    b. Swagger
2. Choose Automation tool/libraries:
    a. Get to know the budget based on that check for tools like playwrite/cypress
        i. Or use libraries: Selenium, RestAsured, BDD, TestNG
        ii. Choose language – Java/Python
        iii. Choose VCS
3. Create a blue print/design work flow of the framework
4. Design it
5. Test it by executing
6. Integrate CI, cross browser tool (based on budget)
7. Integrate reporting library such as extent report/allure

API Test cases format

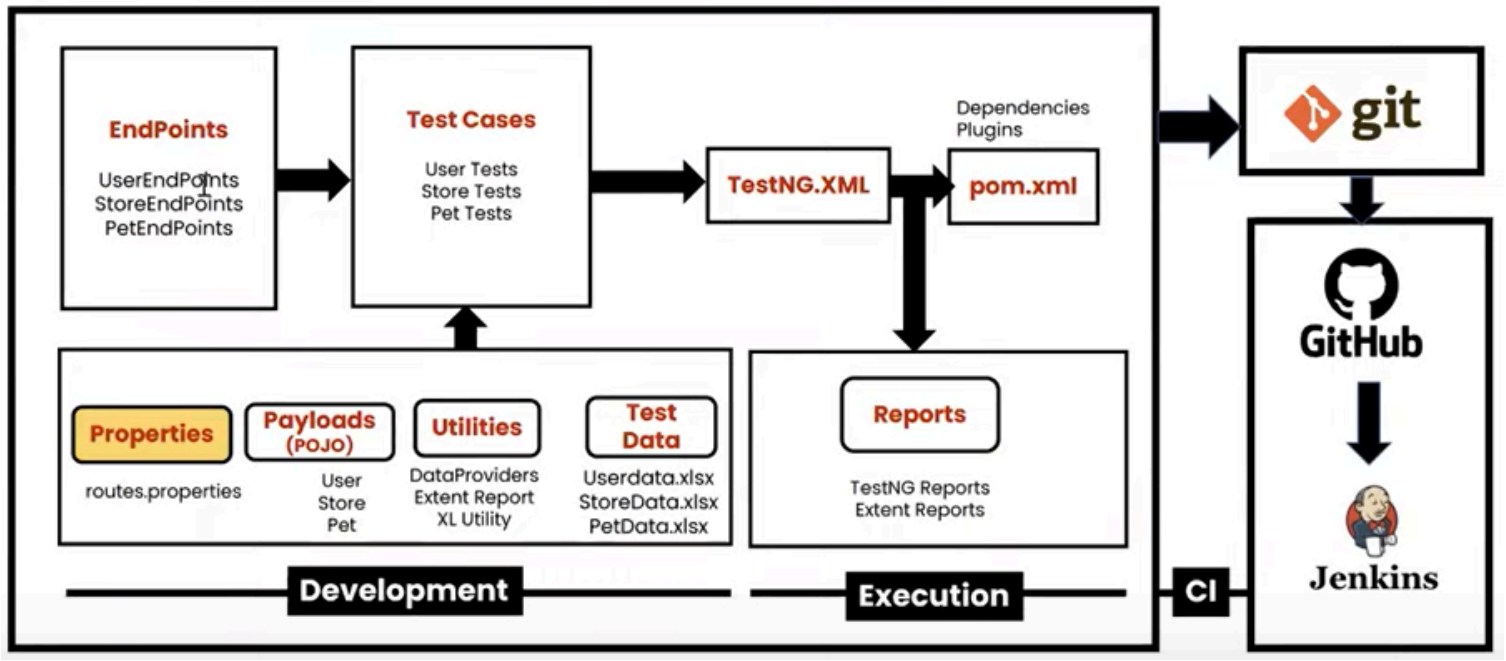
TCID	Model	Title	HTTP Request	URL	Request Body	Response	Authenticaton	Status Code
TC001	User	Create User	Post	ttps://petstore.swagger.io/v2/user	{ "id": 0, "username": "string", "firstName": "string", "lastName": "string", "email": "string", "password": "string", "phone": "string", "userStatus": 0 }	successful operation	NA	200
TC002	User	Get User	Get	https://petstore.swagger.io/v2/user/{username}	Path Param : Username	{ "id": 0, "username": "string", "firstName": "string", "lastName": "string", "email": "string", "password": "string", "phone": "string", "userStatus": 0 }	NA	200
TC003	User	Update User	Put	https://petstore.swagger.io/v2/user/{username}	{ "id": 0,		NA	200

Framework Design workflow at High Level



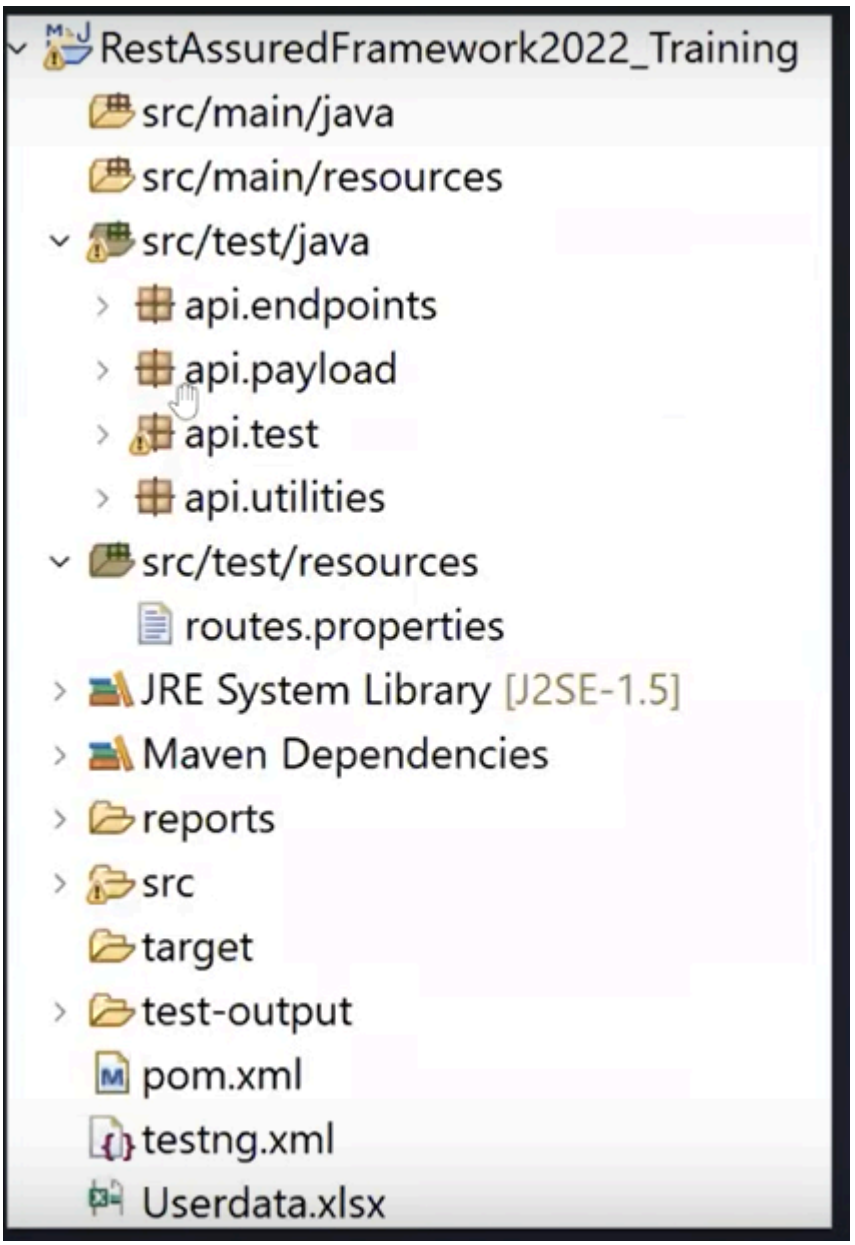
Rest Assured Framework Design

- Routes will be a separate class where we will store all endpoints and send them to test case
- Instead of a separate class for routes some frameworks use properties file



Maven Framework structure

- Src/main are used by developers to add their development code and its resources and src/test are used to add test code and its resources



Data Driven approach

- If we keep data in excel file We will create one excelUtility class and add it in utility package, this class will have methods to get and set data in excel
- Then to pass this data to test case we will use **DataProvider** a testNG annotation to pass the data as param to the test case
- Data provider stores the data from external file in a 2D array

- In excel sheet we store data

	A	B	C	D	E	F	G
1	UserID	UserName	FirstName	LastName	Email	Password	Phone
2	100	ABC	T1	DATA1	abc@gmail.com	test@1	123456789
3	101	DEF	T2	DATA2	def@gmail.com	test@2	987654321
4	102	GHI	T3	DATA3	ghi@gmail.com	test@3	123459876
5							

- We create methods using Apache poi library. Apache poi and apache poi ooxml dependenies are required to read and write the xml file
- 3 main methods we will use:
    - Get row count – it will return total row count
    - Get column count – returns total columns
    - Get cell data – with rowNumber and cellNumber we will get the data from the cell

XLUtility.java | DataProviders.java

```java
/**
  This class contains the methods to read and write the data from and in xml file
  These methods will be used my methods defined in DataProvider class
 */

public FileInputStream fi;
public FileOutputStream fo;
public XSSFWorkbook workbook;
public XSSFSheet sheet;
public XSSFRow row;
public Cell cell;
public CellStyle cellStyle;
String path;

public XLUtility(String path) {
    this.path = path;
}

public int getRowCount(String sheetName) throws IOException {

    fi = new FileInputStream(path);
    workbook = new XSSFWorkbook(fi);
    sheet = workbook.getSheet(sheetName);
    int rowCount = sheet.getLastRowNum();
    workbook.close();
    fi.close();

    return rowCount;
}

public int getCellCount(String sheetName, int rowNum) throws IOException {

    fi = new FileInputStream(path);
    workbook = new XSSFWorkbook(fi);
    sheet = workbook.getSheet(sheetName);
    row = sheet.getRow(rowNum);
    int cellCount = row.getLastCellNum();
    workbook.close();
    fi.close();
```

y.java × DataProviders.java

```java
public int getCellCount(String sheetName, int rowNum) throws IOException {

    fi = new FileInputStream(path);
    workbook = new XSSFWorkbook(fi);
    sheet = workbook.getSheet(sheetName);
    row = sheet.getRow(rowNum);
    int cellCount = row.getLastCellNum();
    workbook.close();
    fi.close();

    return cellCount;
}

public String getCellData(String sheetName, int rowNum, int cellNum) throws IOException {

    fi = new FileInputStream(path);
    workbook = new XSSFWorkbook(fi);
    sheet = workbook.getSheet(sheetName);
    row = sheet.getRow(rowNum);
    cell = row.getCell(cellNum);


    DataFormatter formatter = new DataFormatter();
    String data;

    try {
        // it will get the data at the specified cell in string format even if the data is in other format
        data = formatter.formatCellValue(cell);

    } catch (Exception e) {

        data = "no data found at the specified cell";
    }

    workbook.close();
    fi.close();

    return data;
}
```

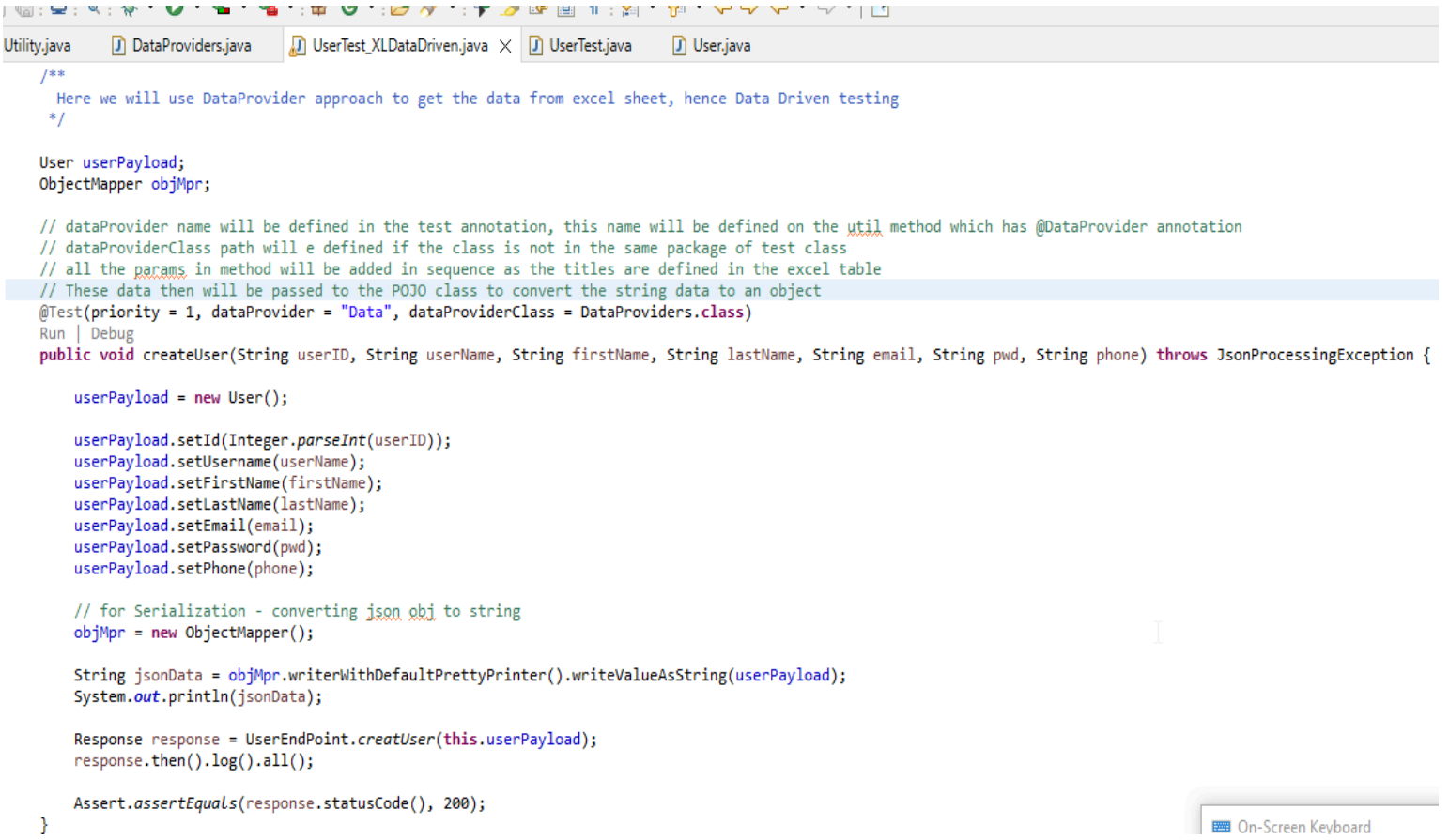
- We will keep these methods in XLUtility class and put it in util package
- We wil create a DataProvider class which will contain method to use above methods to get the data and store it in 2D array if we want data at different cells or use 1D array if we want data at specific fixed cells/column

XLUtility.java | *DataProviders.java

```java
package api.utilities;

import java.io.IOException;

import org.testng.annotations.DataProvider;

public class DataProviders {

    /**
      Here we have the methods with DataProvider annotations which will get the data from the excel
      sheet through XLUtility methods and pass to test case

      DataProviders stores the data in 2D arrays
     */

    @DataProvider(name = "Data")
    public String[][] getAllData() throws IOException{

        String path = System.getProperty("user.dir") + "//test-data//testData.xlsx";
        XLUtility xl = new XLUtility(path);

        int rowNum = xl.getRowCount("Sheet1");
        int cellNum = xl.getCellCount("Sheet1", 1);

        // we store data in 2D array
        String[][] data = new String[rowNum][cellNum];

        // i we take 1 so that it does not read 0th index row which has titles
        for(int i = 1; i<=rowNum; i++) {

            for(int j=0; j<cellNum; j++) {

                // i-1 bcoz we want store data in index position starting from 0 in 2D array
                data[i-1][j] = xl.getCellData("Sheet1", i, j);
            }
        }

        return data;
    }
```

```java
@DataProvider(name = "UserNames")
public String[] getUserNames() throws IOException{

    String path = System.getProperty("user.dir") + "//test-data//testData.xlsx";
    XLUtility xl = new XLUtility(path);

    int rowNum = xl.getRowCount("Sheet1");

    // we store data in 1D array
    String[] data = new String[rowNum];

    // i we take 1 so that it does not read 0th index row which has titles
    for(int i = 1; i<=rowNum; i++) {

        // i-1 bcoz we want store data in index position starting from 0 in 1D array
        data[i-1] = xl.getCellData("Sheet1", i, 1); // we took cell value as 1 bcoz in out excel sheet the username is in column 2
    }

    return data;
}
```

- In test case we will add the DataProvider name and class path(if the dataprovider class is in different package, if its in same package then we don't add class path) with @Test annotation
- dataProvider name will be defined in the test annotation, this name will be defined on the util method which has @DataProvider annotation
- dataProviderClass path will be defined if the class is not in the same package of test class
- all the params in method will be added in sequence as the titles are defined in the excel table
- These data then will be passed to the POJO class to convert the string data to an object

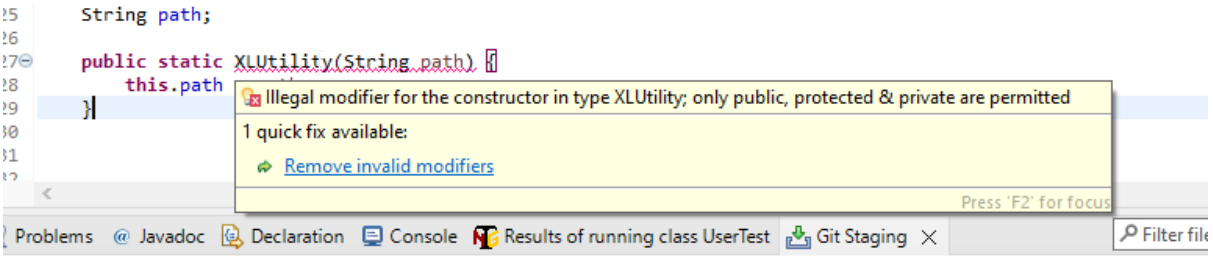


Getting data from Properties file

- We use **ResourceBundle** to get the data from Properties file

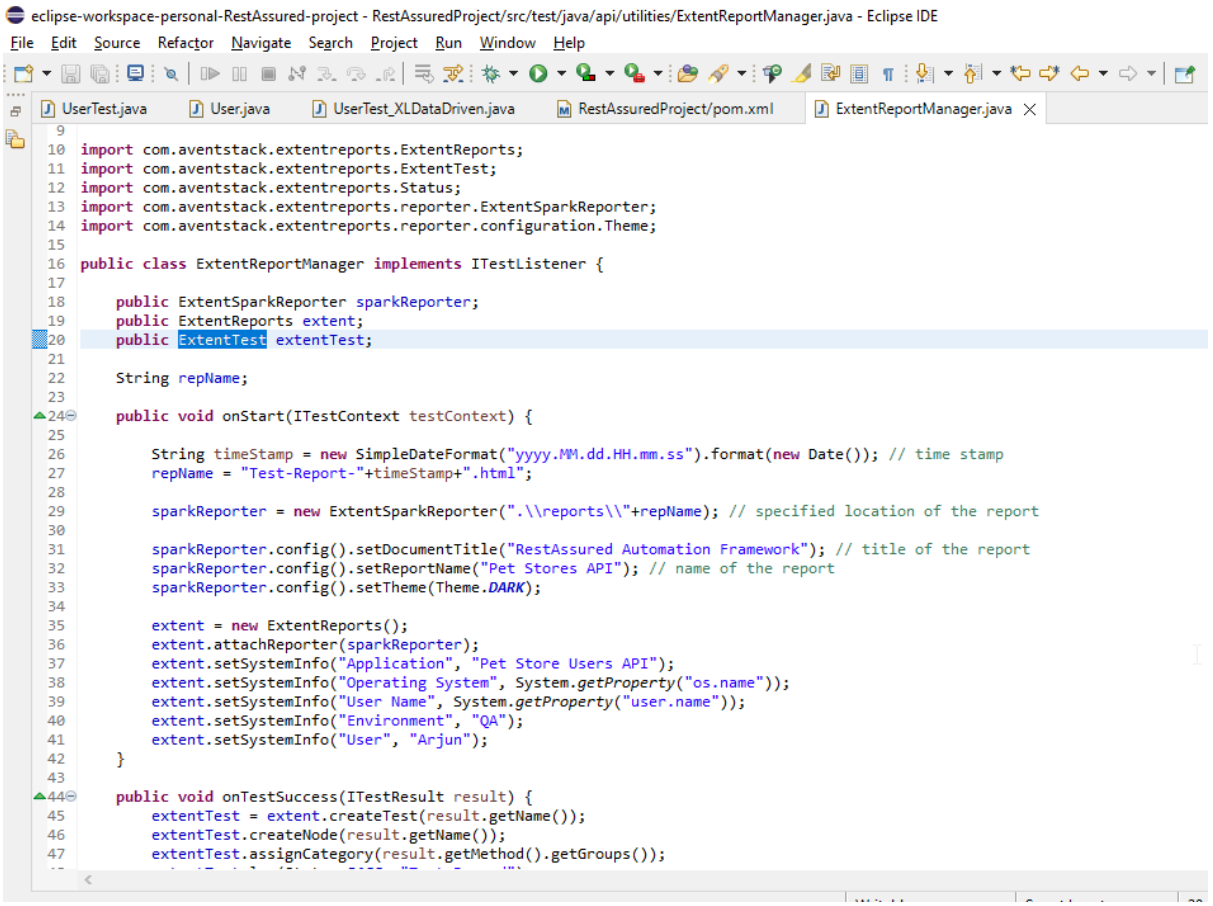
```java
public class UserEndPoints2 {

        // method created for getting URL's from properties file
        static ResourceBundle getURL()
        {
            ResourceBundle routes= ResourceBundle.getBundle("routes"); // Load properties file
            return routes;
        }


        public static Response createUser(User payload)
        {
            String post_url=getURL().getString("post_url");


            Response response=given()
                .contentType(ContentType.JSON)
                .accept(ContentType.JSON)
                .body(payload)
            .when()
                .post(post_url);

            return response;
        }
```

Note: We cannot create a static constructor



Extent Report

- Extent reports avenstack dependency we need to add in pom.xml
- We need to add extent report methods in a separate class lets say extentRepotManager and add it in util package, it will contain extent report objects to create the location, html file, publish data in the file as per the design and format defined
    - **ExtentReports:** its used to instantiate the object and set system properties and add sparkreporter object
    - **ExtentSparkReporter:** this defines the design of the file
    - **ExtentTest:** its used to capture the test logs and add it to the reports
- We use **ITestContext**, **ITestResult** to set report data and get test data and implement **ITestListener** which will be integrated with the testng xml file
- Final method will be **ExtentReports.flush()** this will publish the report
- Now to integrate this to testcase its done through testng xml file



Log4j2 integration

- Add 2 dependencies from - https://logging.apache.org/log4j/2.3.x/maven-artifacts.html

Using Log4j in your Apache Maven build

To build with Apache Maven, add the dependencies listed below to your `pom.xml` file.

pom.xml

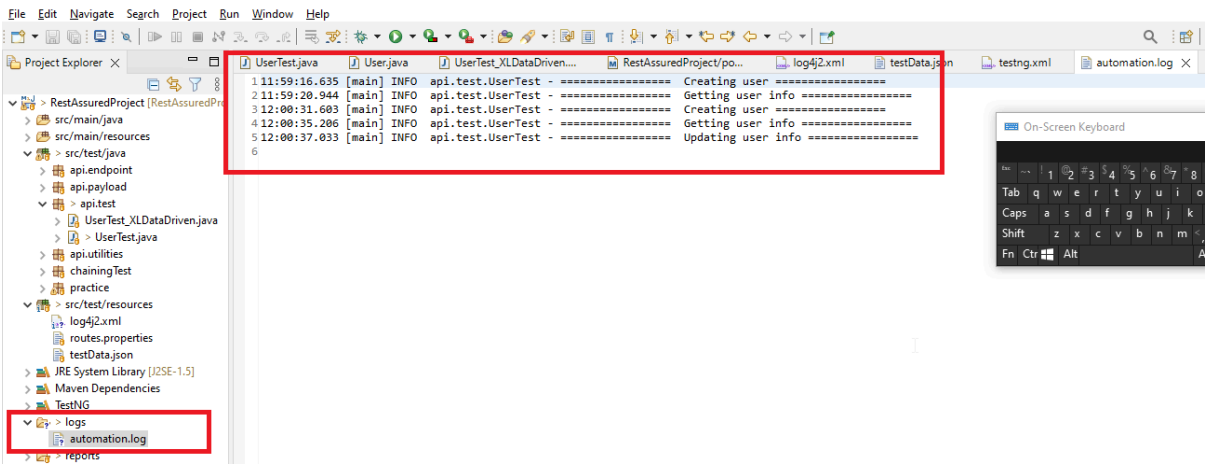
```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.3.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.3.2</version>
  </dependency>
</dependencies>
```

- Now all log configs we will define in log4j2.xml file
- Took the config data from here:
    - https://howtodoinjava.com/log4j2/log4j2-xml-configuration-example/
    - https://logging.apache.org/log4j/2.3.x/manual/configuration.html
- In test class or any util we need to instantiate logger:

```
import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

    Logger logger = LogManager.getLogger(this.getClass());
```

```
17 public class UserTest {
18
19     Faker fk;
20     User userPayload;
21     ObjectMapper objMpr;
22     String jsonData;
23     Logger logger;
24
25     @BeforeClass
26     public void setUp() {
27
28         fk = new Faker();
29
30         userPayload = new User();
31
32         userPayload.setId(fk.idNumber().hashCode());
33         userPayload.setUsername(fk.name().username());
34         userPayload.setFirstName(fk.name().firstName());
35         userPayload.setLastName(fk.name().lastName());
36         userPayload.setEmail(fk.internet().safeEmailAddress());
37         userPayload.setPassword(fk.internet().password(5, 10));
38         userPayload.setPhone(fk.phoneNumber().cellPhone());
39
40         // for Serialization - converting json obj to string
41         objMpr = new ObjectMapper();
42
43         //log4j logger
44         logger = LogManager.getLogger(this.getClass());
45     }
46
```

```java
    }

    @Test(priority = 1)
    Run | Debug
    public void createUser() throws JsonProcessingException {

        logger.info("=================  Creating user =================");

        // Serialization - converting json obj to string
        jsonData = objMpr.writerWithDefaultPrettyPrinter().writeValueAsString(userPayload);
        System.out.println(jsonData);


        Response response = UserEndPoint.creatUser(this.userPayload);
        response.then().log().all();

        Assert.assertEquals(response.statusCode(), 200);
    }

    @Test(priority = 2)
    Run | Debug
    public void getUser() {

        logger.info("=================  Getting user info =================");

        Response response = UserEndPoint.getUser(this.userPayload.getUsername());
        response.then().log().all();

        Assert.assertEquals(response.statusCode(), 200);
    }

```



- We can log debug logs as well, by updating the logger level

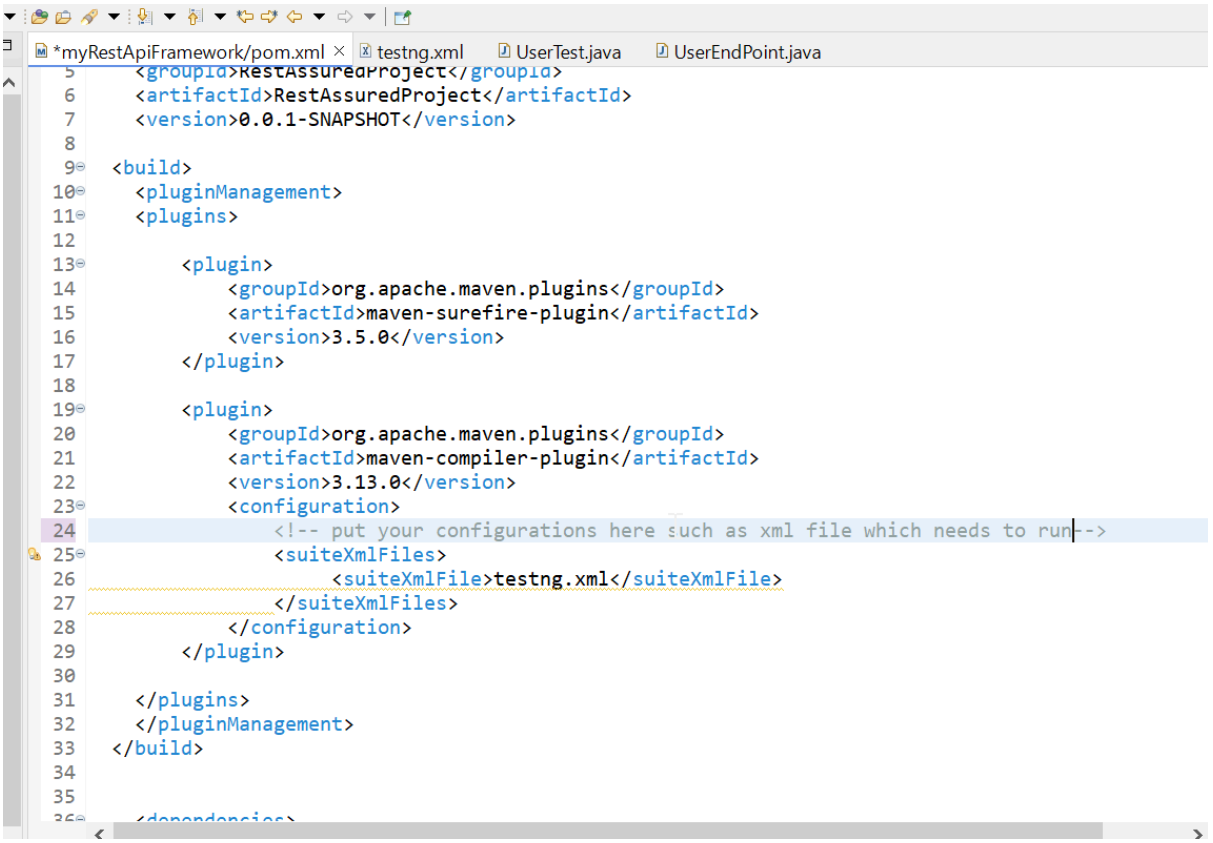
```xml
    </Appenders>

    <Loggers>
        <Root level="debug" >
            <appender-ref ref="File" />
        </Root>
    </Loggers>
</Configuration>
```

Running code from Jenkins and CLI

With Maven project we can run using pom.xml

Run code from POM.XML file

- We need to add 2 plugins:
  - Apache Maven Surefire plugin - https://maven.apache.org/surefire/maven-surefire-plugin/usage.html
  - maven-compiler-plugin - https://maven.apache.org/plugins/maven-compiler-plugin/usage.html
    - Add the testng xml under configuration section in this plugin



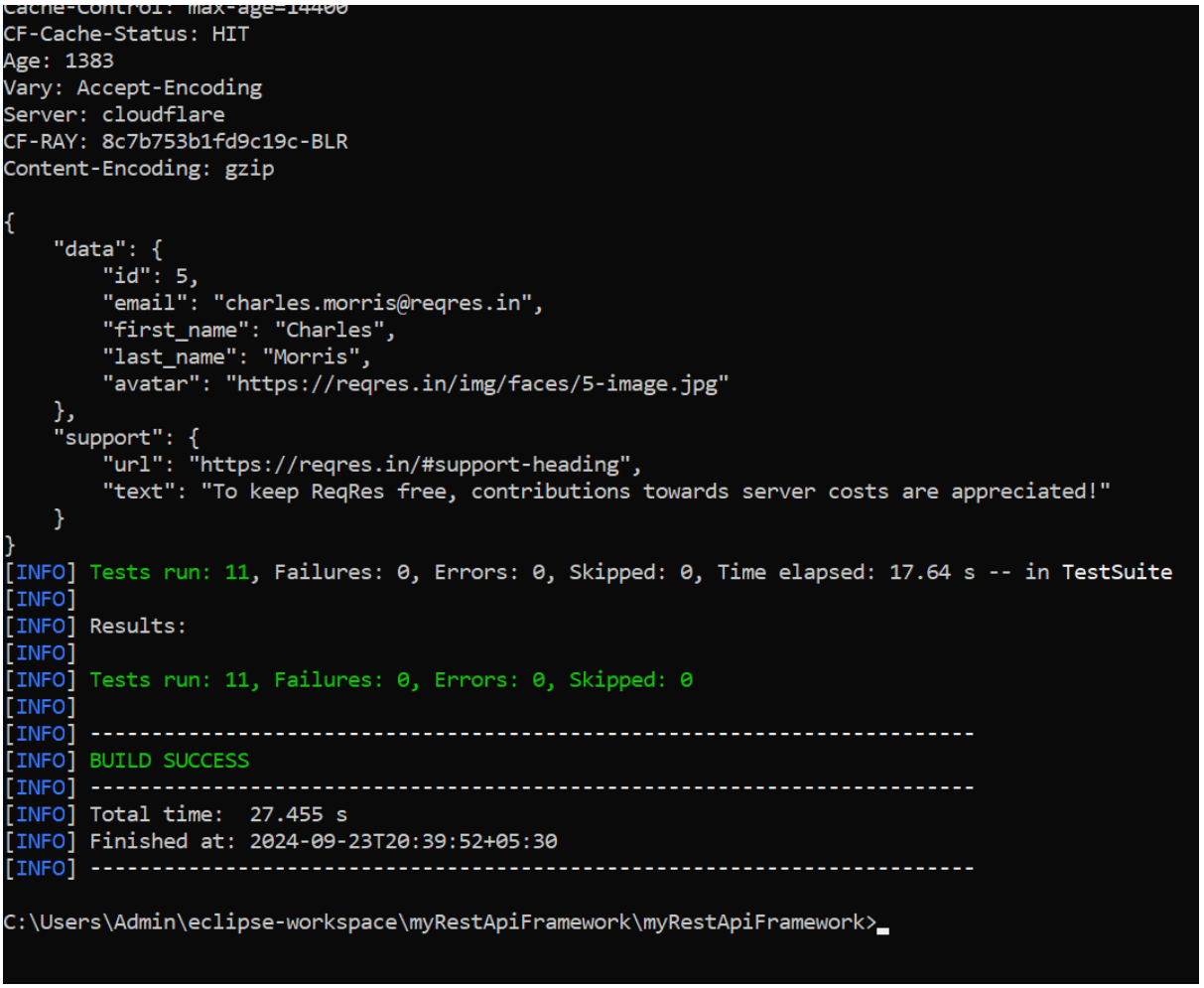
- With this plugins we can run the code via pom.xnl file in any IDE
- To run - Rt click in pom.xml > run > maven test

Run code from CLI

- Outside IDE above plugins wont help us to run the code from CLI for this we need to install Maven in the local machine and add it as a system variable just like Java
- Download here - https://maven.apache.org/download.cgi
- Steps to setup - https://phoenixnap.com/kb/install-maven-windows

Note - We also need java for maven to work

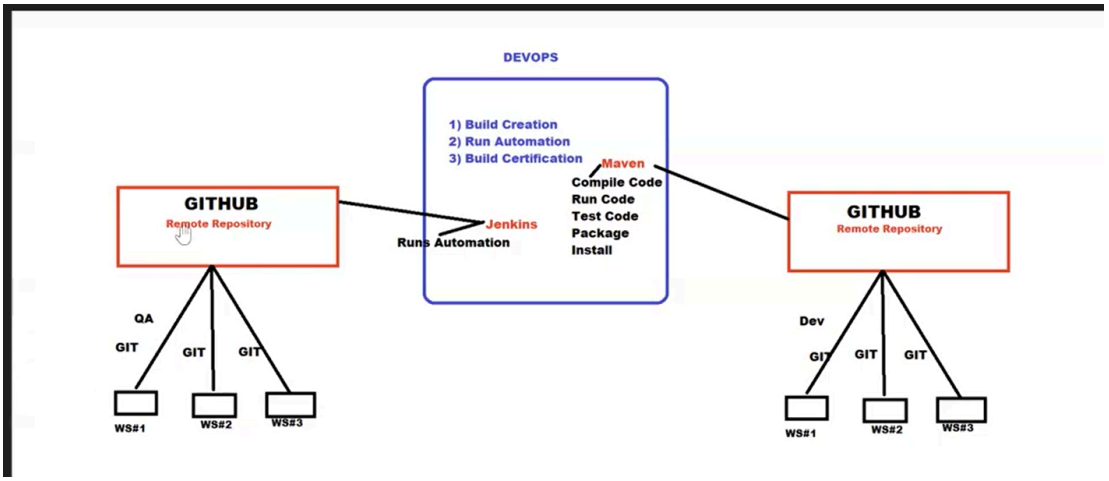
In CMD- From project location enter cmd - **maven clean compile** - to clean and compile the code and enter cmd - **maven test.** This will run the code



Continuous Integration:

- Dev team will develop their code and push it to their local git and from their its pushed to version control system
- Devops team will create a build of the latest code with Maven
- Test team will develop their test scripts and push it to their local git and from their its pushed to version control system
- Devops team will create a build of the latest code in jenkins
- now this test build will run on dev code build
- This complete process is called CI

Note - It is important that code must first run successfully in IDE and then should run successfully in CLI and only after that we can push code to git and from there the correct code will go to jenkins

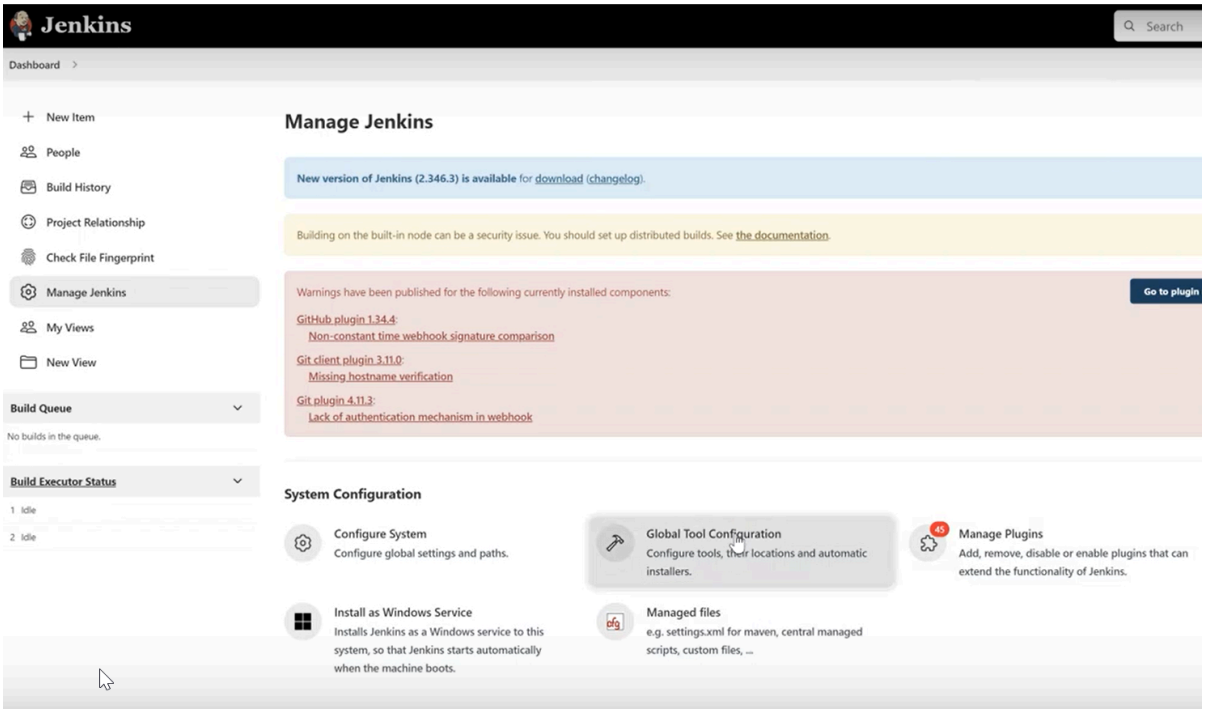


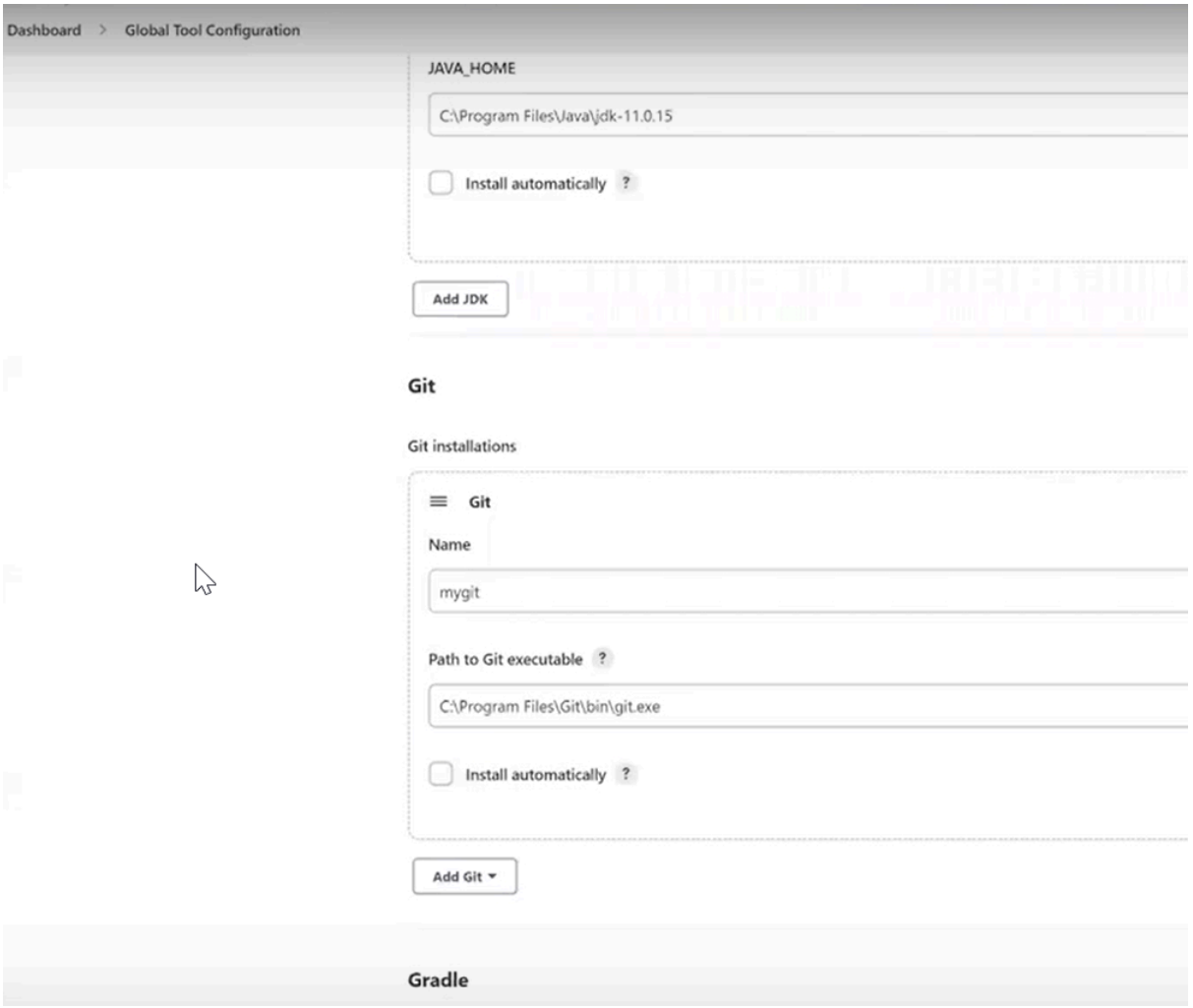
- Once code is present in remote repo - gitHub/bitBucket now its time for us to create a project pipeline in jenkins and integrate the github code there for execution from jenkins

Jenkins setup

- Download and setup process here - https://www.jenkins.io/download/
- jenkins.war cmd in CLI will start the jenkins
- Once jenkins started open the jenkins url on port 8080 and login
- Now double check if the jdk and git path are set correctly in jenkins:

Manage Jenkins > Global Tool configuration





- Now create maven project in jenkins and in its configuration define the git path and pom.xml execution command
- Now the dashboard will have the project and we can run it by clicking on build now